

LIMBAJELE C SI C++ PENTRU INCEPATORI

VOLUMUL II

LIMBAJUL C++



*micro*informatica

LIMBAJELE C SI C++ PENTRU INCEPATORI

VOLUMUL II

LIMBAJUL C++

CLUJ-NAPOCA 1994

Autor
Liviu Negrescu

A confruntat cu originalul
Irina Mitrov

Editura *micro*informatica

Coordonator serie
Smaranda Derveşteanu

Consilier editorial
Andrei Enyedi

Tehnoredactare computerizată
Codruţa Poenaru

Coperta
Liviu Derveşteanu

Tipărit
Imprimeria Ardealul Cluj
Tiraj - 3000 exemplare
Comanda - 376



CUPRINS

20. LIMBAJUL C++ CA EXTENSIE A LIMBAJULUI C

În [14] B. Stroustrup arată că limbajul C++ este proiectat în aşa fel încît:

- să fie un C mai bun;
- să permită stilul de programare prin abstractizarea datelor;
- să permită stilul de programare orientat spre obiecte.

În aceeaşi carte, Stroustrup afirmă că C++ este un "C mai bun" deoarece furnizează un suport mai bun pentru stilurile de programare suportate de limbajul C.

Amintim că limbajul C permite utilizatorului să folosească stilurile de programare încetăţenite sub denumirile de "*programare procedurală*" (vezi 4.14) şi "*programare modulară*" (vezi capitolul 7). Ambele stiluri sînt suportate mai bine de limbajul C++ pe baza introducerii unor extensii la facilităţile existente în limbajul C.

Limbajul C devine un *subset* al limbajului C++ (adesea se obişnuieşte să se spună că C++ este un superset al limbajuluiC). Aceasta înseamnă că un program scris în C este în acelaşi timp şi un program scris în C++.

În linii mari, acest lucru este adevărat. Compatibilitatea nu este asigurată 100% dar cazurile de incompatibilitate sînt neesenţiale şi pot fi uşor eliminate la programe concrete.

În capitolul de faţă se trec în revistă extensiile mai importante ale construcţiilor din C introduse în limbajul C++. Cu alte cuvinte, în acest capitol se discută aspectele care conduc la afirmaţia lui B. Stroustrup că limbajul C++ este un *C mai bun*.

Celelalte facilităţi oferite de limbajul C++ (programarea prin abstractizarea datelor şi orientată spre obiecte) vor fi abordate în alte capitole.

În încheiere, plastic vorbind, putem afirma că limbajul C++ este un "C incrementat" şi înţelegem prin aceasta că pe de o parte este un "C mai bun", fiind o extensie a limbajului C, iar pe de altă parte permite stiluri de programare impracticabile în C:

- programarea prin abstractizarea datelor

şi

– programarea orientată spre obiecte.

20.1. Comentariu

Comentariile sînt explicaţii pentru programatori. Ele pot fi introduse în orice punct în care este legală apariţia unui caracter alb.

În limbajul C, şi deci şi în C++, comentariile se includ între /* (începutul comentariului) şi */ (sfîrşitul comentariului). Comentariile pot fi scrise pe mai multe rînduri.

În afară de această posibilitate, în limbajul C++ există şi varianta de a scrie comentarii pe un *singur* rînd. Un astfel de comentariu începe cu secvenţa:

//

şi se termină la sfîrşitul rîndului.

Comentariile de acest fel însoţesc, de obicei, cîte o instrucţiune sau o componentă a unei instrucţiuni care necesită explicaţii pentru programatori.

Comentariile din antetul unei funcţii se scriu adesea pe mai multe linii şi din această cauză ele se inserează mai simplu folosind convenţia din limbajul C.

Exemplu:

```
...
if(b && zi < 29) //an bisect -> b=1; altfel b=0
  zi++;
...
```

În compunerea unui comentariu se pot utiliza orice caractere ale codului ASCII.

Comentariile sînt eliminate din programe la compilare.

20.2. Definiţia unei funcţii

Definiţia unei funcţii în limbajul C++ se compune din *antet* urmat de *corpul* ei.

O funcţie poate fi apelată dacă este precedată de definiţia sau de prototipul ei.

20.2.1. Antetul şi prototipul unei funcţii

O funcţie în mod implicit este globală. Ea poate fi localizată într-un fişier sursă folosind cuvîntul cheie *static* la începutul antetului ei. În acest caz, se spune că funcţia este *statică*.

O funcţie poate să returneze (întoarcă) sau nu o valoare la revenirea din ea.

Tipul valorii returnate de funcţie se indică în antetul funcţiei. În cazul în care funcţia nu returnează (întoarce) o valoare, în locul tipului din antetul ei se foloseşte cuvîntul cheie *void*.

Antetul unei funcţii în limbajele C şi C++ are formatul:
tip nume(*lista declaraţiilor parametrilor formali*)

sau

static *tip nume* (*lista declaraţiilor parametrilor formali*)

Declaraţiile de parametri sînt separate prin virgule. Dacă funcţia nu are parametri, atunci lista din parantezele rotunde este vidă.

Spre deosebire de limbajul C, în limbajul C++ se pot face iniţializări ale parametrilor formali.

Exemple:

1.
```
Boolean v_calend (int zi,int luna,int an = 1994)
```
În acest exemplu, parametrul formal an este iniţializat cu valoarea 1994.

2.
```
double aria(double r,double a=0,double b=360,
   double pi=3.14159265358979)
```
Parametrii formali iniţializaţi se numesc *parametrii impliciţi*.

La apelul funcţiilor, unui parametru implicit poate să-i corespundă sau nu un parametru efectiv. În cazul în care unui parametru implicit nu-i corespunde, la apel un parametru efectiv, parametrul formal respectiv ia ca valoare valoarea sa iniţială. Aceasta se mai numeşte *valoare implicită* a parametrului respectiv.

Dacă la un apel, unui parametru implicit îi corespunde un parametru efectiv, atunci parametrului formal respectiv i se atribuie în mod obişnuit valoarea parametrului efectiv care îi corespunde. Cu alte cuvinte, în acest caz se neglijează valoarea implicită a parametrului formal.

De exemplu, apelul funcţiei *v_calend* avînd antetul de mai sus:

```
i = v_calend(28,2);
```

implică lansarea funcţiei *v_calend* pentru următoarele valori ale parametrilor formali:

zi = 28, luna = 2, an = 1994.

Pentru *an* s-a utilizat valoarea implicită. În schimb apelul:

```
i = v_calend(31,3,1995);
```

lansează funcţia *v_calend* cu valorile:

zi = 31, luna = 3, an = 1995.

În acest caz, valoarea implicită a fost neglijată.

La utilizarea parametrilor impliciţi există o limitare în legătură cu poziţia acestora şi anume, ei trebuie să ocupe ultimele poziţii în lista declaraţiilor din antetul funcţiei, neputînd fi amestecaţi cu parametri care nu sînt iniţializaţi.

În limbajele C şi C++ se pot defini funcţii cu un număr variabil de parametri. În acest caz,lista declaraţiilor parametrilor formali conţine numai declaraţiile parametrilor care sînt prezenţi la orice apel al funcţiei. Aceştia sînt aşa numiţii *parametri ficşi* ai funcţiei spre deosebire de ceilalţi care se numesc *parametri variabili*. Parametrii ficşi preced pe cei variabili. Prezenţa parametrilor variabili se indică, în antetul funcţiei,prin trei puncte care se scriu după ultimul parametru fix al funcţiei.

Exemplu:

Fie antetul:

```
void vf(int n,double x, ...)
```

Din antet rezultă că funcţia are doi parametri ficşi: *n* şi *x*, precum şi alţi parametri al căror număr şi tip nu este în prealabil precizat şi diferă de la apel la apel.

Funcţiile cu un număr variabil de parametri se definesc folosind nişte macrouri speciale care permit accesul la parametri variabili. Aceste macrouri sînt definite în fişierul *stdarg.h*.

De obicei, funcţiile cu un număr variabil de parametri sînt funcţii de bibliotecă. Exemple de astfel de funcţii din biblioteca sistemului sînt funcţiile *printf* şi *scanf*. Acestea au fiecare un singur parametru fix şi anume parametrul care defineşte formatele de conversie şi eventualele texte constante care intervin în operaţiile de intrare/ieşire iniţiate prin apelul lor.

Antetul unei funcţii poate fi mai general decît cel indicat mai sus. Astfel, în cazul limbajului Turbo C++, antetul unei funcţii are următoarea structură generală:

specificator_de_clasă tip modificatori nume (lista declaraţiilor parametrilor formali)

Specificatorul de clasă, dacă este prezent, este cuvîntul cheie *static* sau *extern*. Cuvîntul cheie *static* se utilizează pentru a localiza funcţia în fişierul în care este definită. Cuvîntul cheie *extern* indică faptul că funcţia este globală. În mod implicit o funcţie se consideră că este globală şi de aceea specificatorul *extern*, de obicei, nu se utilizează.

Tipul din antet defineşte tipul datei returnate de funcţie. Dacă funcţia returnează un alt tip decît cel indicat în antetul ei, atunci valoarea respectivă este convertită automat spre tipul indicat în antet, înainte de a reveni în punctul de după apel.

Dacă tipul nu este prezent în antetul funcţiei, atunci se consideră că funcţia returnează o valoare de tip *int*. Cu toate acestea se recomandă ca tipul să fie prezent totdeauna. S-a constatat că omiterea tipului din antetul funcţiilor conduce la erori, deoarece acesta se omite şi cînd trebuie să fie prezent (adică atunci cînd este diferit de *int*) şi o astfel de eroare nu poate fi depistată de compilator.

Modificatorii care pot apare între tip şi nume sînt specifici diferitelor implementări ale limbajului C++. În cazul limbajului Turbo C++ cei mai importanţi modificatori sînt:

cdecl, pascal, near, far şi huge

Primii doi stabilesc convenţii de transfer a parametrilor şi de apel a funcţiilor. Ei se folosesc cînd se utilizează module scrise în mai multe limbaje de programare.

Modificatorul *cdecl* activează convenţia din C pentru apelul funcţiei respective. În mod analog, modificatorul *pascal* activează convenţia din Pascal pentru apelul funcţiei respective. Convenţia de apel implicită, în lipsa acestor modificatori,este cea definită prin opţiunile de compilare.

Modificatorii *near*, *far* şi *huge* influenţează apelurile funcţiilor. În mod implicit ei se definesc prin modelul de memorie curent şi anume modelele *tiny*, *small* şi *compact* implică modificatorul *near*, modelele *medium* şi *large* implică modificatorul *far*, iar modelul *huge* implică modificatorul *huge*.

Un exemplu de utilizare a modificatorului *far* îl oferă funcţiile de gestiune grafică a ecranului(vezi capitolul 19). Acestea toate au în antet modificatorul *far*, deoarece implică apeluri şi pointeri de tip *far*.

Prototipul unei funcţii poate fi obţinut scriind punct şi virgulă după o construcţie care, fie coincide cu antetul funcţiei respective, fie se obţine din antet eliminînd numele parametrilor formali.

Astfel, prototipul funcţiei *v_calend* al cărui antet se indică în exemplul 1 de mai sus, se poate scrie în următoarele forme:

```
Boolean    v_calend(int    zi,int    luna,
                       int an=1994);
```

sau

```
Boolean v_calend(int,int,int=1994);
```

Prima formă clarifică sensul parametrilor şi de aceea ea este sugestivă în comparaţie cu cea de a doua.

Pentru exemplul 2, se pot utiliza următoarele prototipuri:

```
double aria(double r,double a = 0,double b = 360,
            double pi = 3.14159265358979);
```

sau

```
double aria(double,double = 0,double = 360,
            double = 3.14159265358979);
```

Exerciţii:

20.1 Să se scrie o funcţie care citeşte un întreg scris în baza de numeraţie *b* (2 <= b <= 10), îl converteşte spre tipul *int* şi-l memorează în zona spre care pointează parametrul *n* al funcţiei. Baza implicită a întregului este zece. Funcţia returnează valoarea zero dacă se întîlneşte sfîrşitul de fişier şi unu în caz contrar.

FUNCŢIA BXX1

```
int bint(int *n,int b=10)

/* - citeste un intreg scris in baza b din intervalul [2,10];
     si-l converteste spre int;
   - pastreaza numarul citit in zona spre care pointeaza n;
   - returneaza 0 la sfirsitul de fisier; altfel returneaza 1;
*/

{
 long s;
 int c,sign;
 char er[] =
      "se reia citirea caracterelor intregului\n";

 for(;;){
    printf("tastati caracterele intregului= ");
```

```
    sign = 1;
    while((c = getchar()) == ' ' || c == '\t' ||
        c == '\n')
      ; //salt peste caractere albe
    if(c == '-'){     //numar negativ
      sign = -1;
      c = getchar();    //salt peste minus
    }
    else
      if(c == '+')
        c = getchar();    //salt peste plus
    if(c == EOF)
      return 0;
    if(!(c >= '0' && c < '0'+b)){

    //nu s-a citit o cifra din intervalul [0,b)

      printf("se cer cifre din intervalul\
          [0,%d)\n",b);
      printf(er);
      fflush(stdin);    //se videaza buferul de la intrarea
                        //standard
      continue;
    }
    for(s=0;c >= '0' && c < '0'+b;){
      s = s*b+c-'0';
      if(s > 32767){
        printf("valoarea absoluta a numarului\
            citit depaseste 32767\n");
        printf(er);
        fflush(stdin);
        break;
      }
      c = getchar();
    }
    if(s > 32767) //se reia citirea intregului
      continue;
    *n = sign*s;
    return 1;
  }
}
```

20.2 Să se scrie un program care citeşte o succesiune de numere separate

prin caractere albe şi afişează fiecare număr pe cîte o linie. Numerele citite sînt scrise în sistemul de numeraţie cu baza *b* (2 <= b <= 10). După ultimul număr se tastează sfîrşitul de fişier. Numerele se afişează în sistemul de numeraţie cu baza 10. Baza *b* este argumentul programului din linia de comandă.

PROGRAMUL BXX2

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "BXX1.CPP"

main(int argc,char *argv[])
/* - citeste o succesiune de numere scrise in baza b si le afiseaza in
     sistemul de numeratie cu baza 10;
   - dupa ultimul numar se tasteaza sfirsitul de fisier;
   - 2 <= b <= 10.
*/

{
 int b,lng,n;

 if(argc != 2){
    printf("baza de numeratie nu este argument in\
          linia de comanda\n");
    exit(1);
 }
 lng = strlen(argv[1]);
 if(lng < 1 || lng > 2){
    printf("argument in linia de comanda eronat:\
           %s\n",
    argv[1]);
    exit(1);
 }
 if(argv[1][0] < '0' || argv[1][0] > '9'){
    printf("argumentul din linia de comanda nu\
           este un intreg: %s\n", argv[1]);
    exit(1);
 }
 if(lng == 2 && argv[1][1] != '0'){
    printf("baza de numeratie eronata: %s\n",
          argv[1]);
    exit(1);
```

```
    }
    b = atoi(argv[1]);   //conversia bazei in binar
    if(b < 2 || b > 10){
        printf("baza de numeratie eronata: %s\n",
                argv[1]);
        exit(1);
    }
    for(;;){
        if(b == 10)
            lng = bint(&n);
        else
            lng = bint(&n,b);
        if(lng)
            printf("%d\n",n);
        else
            break;
    }
}
```

20.2.2. Corpul unei funcţii

Corpul unei funcţii este o instrucţiune compusă. În limbajul C,o instrucţiune compusă este o *succesiune de instrucţiuni* care poate fi precedată de o succesiune de declaraţii, succesiuni care sînt incluse între acolade. În particular, o instrucţiune compusă se poate reduce la o pereche de acolade. Acest lucru este adevărat şi în limbajul C++. În plus, în limbajul C++ nu este necesar ca într-o instrucţiune compusă, declaraţiile să preceadă instrucţiunile.

Exemplu:

```
for(int i=1;i < 10;i++){
  ...
}
```

Variabila *i* se declară în antetul ciclului *for*. Această declaraţie este valabilă în continuare pînă la sfîrşitul blocului (instrucţiunii compuse) care conţine instrucţiunea *for* respectivă.

Inserarea declaraţiilor în orice punct al corpului unei funcţii permite să declarăm variabilele locale atunci cînd avem nevoie de ele. În felul acesta se elimină situaţiile în care se declară variabile locale la începutul corpului funcţiei care apoi rămîn neutilizate.

20.3. Tipuri predefinite în C şi C++

Cuvînt cheie	Dimensiune în biţi	Interval
char	8	[-128,127]
signed char	8	[-128,127]
unsigned char	8	[0,255]
enum	16	[-32768,32767]
short	16	[-32768,32767]
int	16	[-32768,32767]
unsigned	16	[0,65535]
long	32	[-2147483648,2147483647]
unsigned long	32	[0,4294967295]
float	32	[3.4*10**(-38),3.4*10**38]
double	64	[1.7*10**(-308),1.7*10**308]
long double	80	[3.4*10**(-4932),1.1*10**4932]
tip *	16 sau 32	
tip near *	16	
tip far *	32	
tip huge *	32	

În cazul datelor flotante (*float*, *double* şi *long double*) se indică intervalele pentru valorile absolute.

Tipul pointer *tip* * se reprezintă pe 16 biţi pentru modelele de memorie *tiny*, *small* şi *compact*. Pentru restul modelelor, datele de tip pointer se

reprezintă pe 32 de biţi.

Datele de tip *float* asigură o precizie de 6-7 cifre, cele de tip *double* o precizie de 15 cifre, iar cele de tip *long double* o precizie de maximum 19 cifre.

20.4. Constante caracter

În limbajulC, o constantă caracter are tipul *int*. Ea se reprezintă printr-un caracter imprimabil inclus între caractere apostrof sau printr-o secvenţă *escape* (vezi 1.6.3.).

În limbajul C++ o astfel de constantă caracter are tipul *char*.

În afară de aceste constante, în limbajul C++ se admit şi constante caracter care au tipul *int*. Ele se reprezintă pe doi octeţi.

Exemple:

'a' - Constantă caracter păstrată în C++ pe un octet.
- `sizeof('a')` are ca valoare pe 1; are tipul *char*.

'ab' - Constantă caracter păstrată pe doi octeţi.
- Octetul mai puţin semnificativ păstrează codul ASCII al lui *a*, iar cel mai semnificativ, codul ASCII al lui *b*.
- `sizeof('ab')` are valoarea 2; are tipul *int*.

'abc' - Eroare.
- Constanta caracter nu poate avea decît cel mult 2 caractere.

Exerciţii:

20.3 Să se scrie un program care afişează dimensiunile în octeţi ale constantelor caracter 'a' şi 'ab', precum şi valorile octeţilor pe care se reprezintă acestea.

PROGRAMUL BXX3

```
#include <stdio.h>

main()
/* - afiseaza dimensiunile in octeti ale constantelor 'a' si 'ab';
   - afiseaza valorile octetilor pe care se reprezinta cele doua constante
     caracter.
*/
{
  printf("dim lui 'a' = %d\n",sizeof 'a');
```

```
   printf("dim lui 'ab' = %d\n",sizeof 'ab');
   printf("codul lui 'a' = %d\n",'a');
   printf("octetul mai putin semnificativ a\
          lui 'ab' = %d\n", 'ab' & 0377);
   printf("octetul mai semnificativ a\
          lui 'ab' = %d\n", ('ab' >> 8) & 0377);
}
```

20.5. Operatori

Operatorii utilizaţi în limbajul C pot fi utilizaţi şi în limbajul C++ şi au aceleaşi priorităţi şi asociativitate.

Tabela cu priorităţile operatorilor din limbajul C este dată în paragraful 3.2.16.

În paragraful de faţă extindem tabela respectivă cu operatori noi, specifici limbajului C++.

20.5.1. Operatorul de rezoluţie

Acest operator permite accesul la o dată globală redefinită ca locală. El se notează prin ::

Fie variabila *i* definită ca globală:

```
int i;
```

şi funcţia *f* în care variabila *i* se redeclară ca şi locală:

```
void f(...)
{
 ...
 char i;
 ...
}
```

În această situaţie, o expresie de forma:

```
i=10;
```

scrisă în corpul funcţiei *f*, atribuie valoarea 10 variabilei locale *i* declarată în corpul funcţiei respective.

Pentru a utiliza variabila globală *i*, în corpul funcţiei *f*, este necesar ca *i* să fie precedată de operatorul de rezoluţie. Cu alte cuvinte, expresia:

```
::i = 10;
```

va atribui valoarea 10, variabilei globale *i.*

Operatorul de rezoluţie este de prioritate maximă, adică are aceeaşi prioritate cu operatorii:

() [] . şi ->

Deci, în tabela operatorilor amintită mai sus,vom completa prima linie cu operatorul de rezoluţie (::).

20.5.2. Operatorul adresă(&)

În limbajul C operatorul adresă se utilizează pentru a defini adresa unei variabile. Astfel, construcţia:

&nume

defineşte adresa de început a zonei de memorie alocată pentru *nume*.

În limbajul C++ acest operator are şi o altă utilizare şi anume aceea de a introduce *tipul referinţă*.

Dacă tipul pointer se introduce prin construcţia:

tip *

tipul referinţă se introduce prin:

tip &

Tipul referinţă creează sinonime pentru nume deja definite. De asemenea, datele de acest tip pot fi folosite pentru a realiza apelul prin referinţă (call by reference).

Avînd în vedere această utilizare nouă a operatorului &, se obişnuieşte ca acesta să mai fie numit şi *operator de referenţiere*.

Operatorul unar * utilizat la declararea pointerilor are în expresii un efect invers celui de referenţiere şi de aceea el se mai numeşte operator de *dereferenţiere*.

Construcţia **tip &** se utilizează în declaraţii de forma:

tip &nume;

Menţionăm că poziţia operatorului & în declaraţia de mai sus este variabilă, adică declaraţia de mai sus poate fi scrisă ca mai jos:

tip & nume;

sau

tip& nume;

Ultima formă este cea mai utilizată.

Numele declarat în acest fel se spune că este o *referinţă.*

O variabilă referinţă se poate iniţializa la definirea sau declararea ei, cu numele unei alte variabile.

Exemplu:

```
int i;

int& j=i;
```

Variabila *j* este sinonimă cu variabila *i.* Ele reprezintă acelaşi întreg.

De exemplu, dacă atribuim lui *i* o valoare:

```
i = 1234;
```

atunci şi *j* are (referă) aceeaşi valoare.

Variabila *j* de mai sus, este un alt nume cu ajutorul căruia avem acces la întregul păstrat în zona de memorie alocată lui *i.*

De aceea, expresiile:

```
i*7+3
```

şi

```
j*7+3
```

au aceeaşi valoare.

De asemenea, atribuirea:

```
j=12345
```

păstrează întregul 12345 în zona de memorie alocată pentru *i.*

Declaraţia:

```
int & j=i;
```

trebuie interpretată ca fiind procedeul prin care *j* se defineşte a fi un nume sinonim pentru *i.* Acest mod de interpretare a declaraţiei de mai sus diferă de sensul obişnuit al iniţializărilor care se fac prin declaraţii.

Variabilele referinţă se utilizează frecvent pentru a avea acces la zonele de memorie alocate dinamic în memoria heap.

20.5.3. Operatorul de alocare dinamică a memoriei (new)

În limbajul C se pot aloca zone de memorie în memoria *heap* folosind

funcţii de bibliotecă. Astfel de funcţii sînt, de exemplu, funcţiile *malloc* şi *calloc* (vezi 8.8).

În afara acestor funcţii, limbajul C++ permite alocări în zona *heap* prin intermediul operatorului *new*. Acesta este unar şi are aceeaşi prioritate ca şi ceilalţi operatori unari.

Operatorul *new* are ca valoare adresa de început a zonei de memorie alocată în memoria *heap* sau zero (pointerul nul) în cazul în care nu se poate face alocarea.

Operandul operatorului *new* în cea mai simplă formă, este *numele unui tip* (predefinit sau definit de utilizator).

Exemple:

1.
```
int *pint;
pint = new int;
```

Prin intermediul acestei expresii, se alocă în memoria *heap*, o zonă de memorie în care se pot păstra date de tip *int*.

Adresa de început a zonei alocate se atribuie pointerului *pint*. Expresia:

```
*pint = 100;
```

păstrează întregul 100 în zona respectivă.

Aceeaşi alocare se obţine, în limbajul C, folosind funcţia *malloc* ca mai jos:

```
pint = (int *)malloc(sizeof(int));
```

2.
```
int& i = *new int;
```

Prin intermediul acestei declaraţii (definiţii) se alocă în memoria *heap* o zonă de memorie în care se pot păstra date de tip *int*. Numele *i* permite referirea la întregul păstrat în zona respectivă. Expresia de atribuire:

```
i = 100
```

păstrează întregul 100 în zona respectivă.

Zonele de memorie alocate cu ajutorul operatorului *new* pot fi iniţializate.

În acest scop se utilizează o expresie de forma:

new *tip*(*expresie*)

unde:

tip - Este numele unui tip.

expresie - Este o expresie a cărei valoare iniţializează zona de memorie alocată prin operatorul *new*.

Exemple:

1.
```
double *pdouble;
```
```
pdouble = new double(3.14159265);
```
Această instrucţiune realizează următoarele:

- alocă în memoria *heap* o zonă de memorie în care se păstrează valoarea 3.14159265 în flotantă dublă precizie;
- adresa de început a acestei zone de memorie se atribuie variabilei *pdouble*.

2.
```
double& pi = *new double(3.14159265);
```
Prin această declaraţie se rezervă, în memoria *heap*, o zonă de memorie în care se păstrează valoarea 3.14159265 în flotantă dublă precizie. Data respectivă se poate referi cu ajutorul numelui *pi*. De exemplu, *pi* poate fi utilizat în mod obişnuit în expresii de forma:
```
pi*r*r
sin(pi/2)
x*180/pi
```
etc.

O altă utilizare importantă este aceea de alocare a unei zone de memorie, în memoria *heap*, pentru tablouri.

În acest scop, utilizăm o expresie de forma:

new *tip[exp]*
unde:

exp - Este o expresie de tip întreg.

Prin această construcţie se rezervă, în memoria *heap*, o zonă de memorie de *exp* ***sizeof**(*tip*) octeţi.

Valoarea expresiei de mai sus, este adresa de început a zonei de memorie rezervată prin operatorul *new*.

Exemplu:

```
double *tab;
int m,n;
...
```

```
tab=new double[m*n];
```

Această instrucţiune rezervă în memoria *heap* o zonă de *m*n****sizeof(double)** octeţi. Adresa de început a acestei zone de memorie se atribuie pointerului *tab*.

Elementele unei astfel de zone de memorie nu pot fi inţializate decît numai prin secvenţe de program corespunzătoare. De exemplu, pentru a anula cele *m*n* elemente de tip *double* rezervate ca mai sus, putem folosi instrucţiunea *for* de mai jos:

```
for(int i=0; i < m*n;i++)
  tab[i]=0.0;
```

20.5.4. Operatorul de dezalocare a memoriei (delete)

O zonă de memorie alocată prin operatorul *new* se eliberează prin operatorul *delete*.

Dacă *p* este un pointer spre *tip*:

*tip *p*;

şi

p = **new** *tip*;

atunci zona din memoria *heap* alocată cu ajutorul lui *new* se eliberează folosind construcţia:

delete *p*;

De asemenea, operatorul *delete* se utilizează pentru a dezaloca tablourile alocate prin *new*.

Fie alocarea:

*tip *p* = **new** *tip*[*expresie*];

Această zonă se eliberează folosind o construcţie de forma:

delete [*expresie*] *p*;

Menţionăm că expresia care defineşte numărul elementelor tabloului nu este totdeauna necesară la dezalocare.

Mai tîrziu, vom reveni asupra operatorilor *new* şi *delete* şi vom indica situaţia cînd este necesară prezenţa dimensiunii tabloului la dezalocarea lui.

La utilizarea operatorului *delete* este important ca pointerul la care se aplică să aibă o valoare obţinută ca rezultat al aplicării prealabile a operatorului *new*.

Dacă operatorul *delete* se aplică la un pointer nenul a cărui valoare nu a fost obţinută ca rezultat al aplicării operatorului *new*, atunci efectul aplicării lui *delete* este imprevizibil.

20.6. Apel prin referinţă (call by reference)

În limbajul C apelul se face prin valoare (call by value). Acesta devine prin referinţă cînd parametrul efectiv este un nume de tablou (vezi 4.15.). De asemenea, apelul prin referinţă se poate realiza cu ajutorul pointerilor (vezi 8.2). În capitolul 8 sînt date o serie de exerciţii care utilizează pointerii pentru a realiza apelul prin referinţă (ex. 8.1, 8.2, 8.3, 8.6, 8.7).

În limbajul C++ se pot utiliza toate facilităţile amintite mai sus cu privire la apelul funcţiilor. În plus, s-a introdus apelul prin referinţă. Acesta se realizează prin intermediul parametrilor de tip referinţă.

Exemplul devenit deja clasic pentru explicarea apelului prin referinţă este cel al funcţiei de permutare a valorilor a două variabile.

Fie funcţia *perm* definită ca mai jos:

```
void perm(int x,int y)
{
 int t;

 t = x;
 x = y;
 y = t;
}
```

Această funcţie nu are nici un efect asupra parametrilor efectivi de la apelurile ei.

Într-adevăr, fie apelul:

```
int a,b;
...
perm(a,b);
...
```

Apelul fiind prin valoare, se atribuie parametrilor formali *x* şi *y* valorile lui *a* şi respectiv *b*. Funcţia permută valorile parametrilor formali *x* şi *y* dar această permutare nu are nici un efect asupra parametrilor efectivi *a* şi *b*.

Pentru ca funcţia să permute valorile parametrilor efectivi este necesar să se realizeze apelul ambilor parametri prin referinţă.

În limbajul C (şi deci şi în limbajul C++) este posibil să realizăm acest

lucru folosind pointerii, ca mai jos:

```
void pperm( int *x,int *y)
{
 int t;

 t = *x;
 *x = *y;
 *y = t;
}
```

În acest caz, funcţia *pperm* se apelează astfel:

```
int a,b;
...
pperm(&a,&b);
```

Prin acest apel se atribuie pointerilor *x* şi *y* adresele parametrilor *a* şi respectiv *b*. Funcţia *pperm* nu permută valorile pointerilor *x* şi *y* ci valorile întregilor spre care pointează cei doi pointeri, adică chiar valorile lui *a* şi *b*.

În limbajul C++ se poate defini funcţia de permutare cu parametri formali de tip referinţă:

```
void rperm(int& x,int& y)
{
 int t;

 t = x;
 x = y;
 y = t;
}
```

Funcţia *rperm* se apelează astfel:

```
int a,b;
...
rperm(a,b);
...
```

În acest caz *x* şi *y* sînt sinonime cu variabilele *a* şi respectiv *b*. Aceasta înseamnă că *x* şi *y* accesează aceleaşi date din memorie ca şi *a* şi *b*. De aceea, permutarea valorilor referite de *x* şi *y* înseamnă permutarea valorilor referite de *a* şi *b*.

Comparînd funcţiile *perm* şi *rperm* observăm că ele se apelează la fel şi singura diferenţă dintre ele constă în modul de declarare al parametrilor formali. În cazul funcţiei *perm* parametrii formali sînt date de tip *int*, iar în

cazul funcţiei *rperm* aceştia sînt referinţe la date de tip *int*.

Exerciţii:

20.4 Să se scrie o funcţie care afişează caracterele unui tablou şi citeşte un întreg de tip *int*. Funcţia are doi parametri:

text: - Tablou unidimensional de tip caracter şi care are aceeaşi utilizare ca în exerciţiul 6.3.

x: - Parametru de tip referinţă la întregi de tip *int*.
- Numărul citit se păstrează în zona de memorie alocată parametrului efectiv corespunzător lui *x*.

Funcţia returnează valoarea zero la întîlnirea sfîrşitului de fişier şi unu în caz contrar.

Această funcţie este analogă cu funcţiile definite în exerciţiile 6.3 şi 8.2.

În cazul funcţiei din exerciţiul 6.3, numărul citit se atribuie variabilei globale *v_int*. Funcţia definită în exerciţiul 8.2., păstrează numărul citit în zona de memorie spre care pointează parametrul formal *x* de tip pointer.

Cu alte cuvinte, aceste funcţii sînt 3 variante pentru a citi un întreg de tip *int* după ce, în prealabil, ele afişează un şir de caractere. Funcţia definită în exerciţiul 6.3. este varianta care utilizează o *variabilă globală*, cea din exerciţiul 8.2. este variantă cu *pointer*, iar funcţia definită mai jos este variantă cu parametru de tip referinţă (apel prin referinţă).

FUNCŢIA BXX4

```
int rcit_int(char text[],int& x)

/* - afiseaza sirul de caractere spre care pointeaza text;
   - citeste un intreg si-l pastreaza in zona de memorie alocata
     parametrului efectiv corespunzator lui x;
   - returneaza:
          0 - la intilnirea sfirsitului de fisier;
          1 - altfel.
*/
{
 char t[255];

 for(;;){
      printf(text);
      if(gets(t)==0)
          return 0;
      if(sscanf(t,"%d",&x)==1)
```

```
        return 1;
    }
}
```

Observaţie:

Funcţia de faţă se apelează prin apeluri de forma:

```
int n;
...
if(rcit_int("n=",n)){
...
}else       // s-a tastat EOF
...
```

Parametrul *x* este sinonim cu *n* şi de aceea la apelul funcţiei *sscanf*, expresia *&x* înseamnă *&n*.

20.5 Să se scrie o funcţie care citeşte un întreg de tip *int* care aparţine unui interval dat.

Această funcţie este analogă cu funcţia definită în exerciţiul 8.3. Ea foloseşte un parametru de tip referinţă în locul parametrului de tip pointer *pint* al funcţiei din exerciţiul amintit mai sus.

FUNCŢIA BXX5

```
int rcit_int_lim(char text[],int inf,int sup,
                 int & rint)

/* - afiseaza sirul de caractere spre care pointeaza text;
   - citeste un intreg de tip int ce apartine intervalului [inf,sup] si-l
     pastreaza in zona de memorie alocata parametrului efectiv
     corespunzator lui rint;
   - returneaza:
        0 - la intilnirea sfirsitului de fisier;
        1 - altfel.
*/
{
 for(;;){
    if(rcit_int(text,rint)==0)
        return 0;   // s-a intilnit EOF
    if(rint >= inf&&rint <= sup)
        return 1;
    printf("intregul tastat nu apartine\
            intervalului:");
    printf("[%d,%d]\n",inf,sup);
```

```
        printf("se reia citirea\n");
    }
  }
```

20.6 Să se scrie o funcţie care citeşte o dată calendaristică compusă din *zi*, *luna* şi *an*. Funcţia validează data calendaristică respectivă.

Această funcţie este analogă cu funcţia definită în exerciţiul 8.4. Deosebirea dintre ele constă în aceea că parametrii funcţiei din exerciţiul 8.4. sînt de tip pointer, iar cei din exerciţiul de faţă sînt de tip referinţă.

FUNCŢIA BXX6

```
int rcit_data_calend(int& rzi,int& rluna,int& ran)
/* - citeste data calendaristica, o valideaza si o pastreaza in zonele de
     memorie alocate parametrilor efectivi de la apel;
   - returneaza:
         0 - la intilnirea sfirsitului de fisier;
         1 - altfel.
*/
{
  static char ziua[] = "ziua:";
  static char luna[] = "luna:";
  static char an[]   = "an:";
  static char er[]   = "s-a tastat EOF";

  for(;;){    // se citeste ziua, luna si anul
    if(rcit_int_lim(ziua,1,31,rzi)==0){
       printf("%s\n",er);
       return 0;
    }
    if(rcit_int_lim(luna,1,12,rluna)==0){
       printf("%s\n",er);
       return 0;
    }
    if(rcit_int_lim(an,1600,4900,ran)==0){
       printf("%s\n:",er);
       return 0;
    }

    //validare data calendaristica
    if(v_calend(rzi,rluna,ran))
       return 1;
    printf("data calendaristica este eronata\n");
```

```
        printf("se reia citirea datei\
                calendaristice\n");
    }
}
```

20.7 Să se modifice funcţia din exerciţiul 8.1, înlocuind parametrii de tip pointer cu parametrii de tip referinţă.

FUNCŢIA BXX7

```
void rluna_si_ziua(int zz,int an,
                   int & zi,int& luna)

/* determina luna si ziua din luna;
      zz  - este ziua din an;
      an  - este anul calendaristic;
      zi  - referinta la zona in care se pastreaza ziua din luna;
      luna - referinta la zona in care se pastreaza luna.
*/
{
 int bisect = an %4==0&&an%100||an%400==0;
 int i;
 extern int nrzile[];

 for(i=1; zz>nrzile[i]+(i==2&&bisect); i++)
          zz -= (nrzile[i]+(i==2&&bisect));
 zi = zz;
 luna = i;
}
```

20.8 Să se scrie un program care citeşte o dată calendaristică şi afişează data calendaristică pentru ziua următoare.

Acest program este analog cu cele definite în exerciţiile 6.9. şi 8.5.

Programul de faţă foloseşte funcţii care utilizează parametri de tip referinţă pentru interfaţă în locul variabilelor globale sau al parametrilor de tip pointer.

PROGRAMUL BXX8

```
#include <stdio.h>
#include <stdlib.h>
#include "bxx4.cpp"  //functia rcit_int
#include "bxx5.cpp"  //functia rcit_int_lim
#include "bvi5.cpp"  //functia v_calend
```

```
#include "bvi6.cpp"  // functia zi-din-an
#include "bxx6.cpp"  // functia rcit_data_calend
#include "bxx7.cpp"  // functia rluna_si_ziua

int nrzile[] =
        {0,31,28,31,30,31,30,31,31,30,31,30,31};

main()
/* citeste o data calendaristica, o valideaza si in caz ca este corecta,
   afiseaza data calendaristica a zilei urmatoare */

{
 int zz,ll,aa;

 // citeste si valideaza data calendaristica
 if(rcit_data_calend(zz,ll,aa)==0)
     exit(1);
 if(zz==31&&ll==12){

 /* 31 decembrie; ziua urmatoare este 1 ianuarie din anul urmator */
     zz=1;   // 1 ianuarie
     ll=1;
     aa++;   // anul urmator
 }
 else        // se determina ziua urmatoare
     rluna_si_ziua(zi_din_an(zz,ll,aa)+1,aa,zz,ll);

 // afiseaza data calendaristica a zilei urmatoare
 printf("ziua:%d\tluna:%d\tan:%d\n",zz,ll,aa);
}
```

20.9 Să se scrie o funcţie care citeşte:

- valoarea variabilei *m* de tip *int*;
- valoarea variabilei *n* de tip *int*;
- *m*n* numere care reprezintă elementele unei matrice de ordinul *m*n*.

Funcţia de faţă este analogă cu funcţia definită în exerciţiul 8.6. Ea foloseşte parametri de tip referinţă în locul parametrilor de tip pointer.

FUNCŢIA BXX9

```
int rdcitmat(double dmat[],int max,
             int& nrlin,int& nrcol)

/* - citeste pe m   - numar de linii;
                n   - numar de coloane;
                m*n - numere de tip double pe care le pastreaza
     in matricea dmat prin liniarizare;
   - returneaza valoarea m*n;
   - valoarea lui m se pastreaza in zona referentiata de nrlin;
   - valoarea lui n se pastreaza in zona referentiata de nrcol.
*/
{
 int i;
 char t[255];
 char er[] = "s-a tastat EOF\n";

 do{     // se citesc valorile lui m si n
     // citeste pe m
     if(rcit_int_lim("numarul de linii=",1,
        max,nrlin)==0){
          printf(er);
          exit(1);
     }

     // citeste pe n
     if(rcit_int_lim("numarul de coloane=",1,
        max,nrcol)==0){
          printf(er);
          exit(1);
     }
     i=nrlin*nrcol;
     if(i <= max)
          break;
     printf("produsul m*n = %d depaseste pe\
               max=%d\n",i,max);
     printf("se reiau citirile lui m si n\n");
  }while (1);

  /* se citesc cele m*n elemente ale matricei tastate pe linii */
  if(ndcit(i,dmat) != i){
     printf("nu s-au tastat %d elemente\n",i);
     exit(1);
```

```
  }
 return i;
}
```

20.10 Să se scrie un program care citeşte elementele de tip *double* ale două matrice *a* şi *b*, calculează produsul lor şi-l afişează.

Acest program este analog cu cel din exerciţiul 8.7.

În cazul programului de faţă se folosesc funcţii cu parametri de tip referinţă în locul funcţiilor cu parametri de tip pointer.

PROGRAMUL BXX10

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "bxx4.cpp"   //rcit_int
#include "bxx5.cpp"   //rcit_int_lim
#include "biv37.cpp"  //ndcit
#include "bxx9.cpp"   //rdcitmat

#define MAX 900

main() /* citeste elementele a doua matrice, calculeaza s i
          afiseaza matricea produs, cite 4 elemente pe o linie */
{
 int i,j,k;
 int m,n,p,s,r,q;
 int mn,np;

 double a[MAX],b[MAX],c[MAX];

 //citeste matricea a
 mn=rdcitmat(a,MAX,m,n);

 //citeste matricea b
 np=rdcitmat(b,MAX,p,s);
 if(n!=p){
    printf("numarul coloanelor matricei\
          a=%d\n",n);
    printf("difera de numarul liniilor ");
    printf("matricei b=%d\n",p);
    exit(1);
 }
```

```
// se calculeaza produsul c=a*b
for(i=0;i<m;i++){
    q=i*n;
    for(j=0;j<s;j++){
        r=i*s+j;
        c[r]=0.0;
        for(k=0;k<n;k++)
        c[r] += a[q+k]*b[k*s+j];
    }
}

// afiseaza elementele matricei produs
printf("\n\n\t\t matricea produs\n");
k=1;
for(i=0;i<m;i++){
    p = i*s;
    for(j=0;j<s;j++){
        printf("c[%d,%d]=%8g ",i,j,c[p+j]);
        if(j%4==3){

            // afiseaza 4 elemente pe un rind
            printf("\n");
            k++; // numara liniile
        }
        if(k==23){
            printf("actionati o tasta pentru a\
                      continua\n");
            getch();
            k=1;
        }
    }
    printf("\n");
    k++;
}
}
```

20.11 Să se scrie o funcţie care păstrează un şir de caractere într-o zonă de memorie alocată în memoria *heap* cu ajutorul operatorului *new*. Funcţia returnează adresa de început a zonei în care se păstrează şirul de caractere sau zero (pointerul nul), în cazul în care nu se poate rezerva zona respectivă în memoria *heap*.

Această funcţie este analogă cu funcţia din exerciţiul 8.15. Aceasta din urmă utilizează funcţia *malloc* pentru a aloca zona de memorie *heap*.

FUNCŢIA BXX11

```
char *nmemsir(char *s)
```

/* pastreaza in memoria heap sirul de caractere spre care pointeaza s */

```
{
 char *p;

 if((p=new char[strlen(s)+1]) !=0){
```

/* p are ca valoare adresa de inceput a zonei de memorie rezervata in memoria heap si care are strlen(s)+1 octeti */

```
     strcpy(p,s);   // copierea sirului in memoria heap
     return p;
 }else
     return 0;   // nu s-a putut rezerva zona in memoria heap
}
```

20.12 Să se scrie un program care citeşte o succesiune de cuvinte şi-l afişează pe cel mai mare.

Acest program este analog cu cel din exerciţiul 8.16. În cazul de faţă, se utilizează operatorul **delete** pentru a elibera o zonă de memorie din zona **heap** spre deosebire de programul din exerciţiul 8.16. care utilizează în acest scop funcţia **free**.

PROGRAMUL BXX12

```
#include <stdio.h>
#include <string.h>
#include "bxx11.cpp"  // nmemsir

#define MAX 100

main()
/* citeste o succesiune de cuvinte si-l afiseaza pe cel mai mare */
{
 char cuvcrt[MAX+1];
 char *cuvmax=0;  // pointeaza spre cuvintul maxim

 while(scanf("%100s",cuvcrt)!=EOF)
     if(cuvmax==0)  // prima citire
        cuvmax=nmemsir(cuvcrt);
     else
```

```
        if(strcmp(cuvcrt,cuvmax)>0){
                /* cuvintul citit este mai mare decit cel din memoria
                   heap si spre care pointeaza cuvmax */
               delete cuvmax; // se elibereaza zona din memoria
                                     // heap
               // se pastreaza cuvintul citit curent in memoria heap
               cuvmax=nmemsir(cuvcrt);
        }
  printf("cel mai mare cuvint este\n");
  printf("%s\n",cuvmax);
 }
```

20.7. Funcţii care returnează date de tip referinţă

În limbajul C++, o funcţie poate returna o referinţă la o dată de un tip oarecare. În acest caz, *tip* din antetul unei funcţii are forma:

tip&

Exemplu:

```
typedef struct {
 double x;
 double y;
} COMPLEX;
COMPLEX z;
...
double& re = retreal(z);
double& im = retimag(z);
```

Funcţiile *retreal* şi *retimag* returnează referinţă la partea reală şi respectiv cea imaginară a numărului complex *z*. Ele se definesc ca mai jos:

```
double& retreal(COMPLEX& c)
{
 return c.x;
}
double& retimag(COMPLEX& c)
{
 return c.y;
}
```

O funcţie care returnează o referinţă poate fi apelată atît în dreapta semnului egal cît şi în stînga. Un astfel de apel este un operand atît *lvalue* cît

şi *rvalue* (vezi 8.7.).

De exemplu, funcţiile *retreal* şi *retimag* pot fi utilizate pentru a iniţializa numărul complex *z* din exemplul de mai sus, scriind:

```
retreal(z) = 1;
retimag(z) = -3;
```

Prima instrucţiune are acelaşi efect cu atribuirea:

```
z.x = 1;
```

iar cea de a doua cu atribuirea:

```
z.y = -3;
```

Evident, atribuirile de forma z.x = ... şi z.y = ... sînt operaţii simple care nu se justifică a fi înlocuite prin atribuiri în care se apelează funcţiile *retreal* şi *retimag*.

Astfel de apeluri se utilizează în cazul în care funcţiile apelate reprezintă procese de calcul mai complexe.

Apelurile funcţiilor care returnează referinţă, fiind operanzi de tip *lvalue*, lor li se pot aplica operatorii de incrementare şi decrementare. Astfel, instrucţiunile:

```
++retreal(z);
```

şi

```
retreal(z)++;
```

sînt corecte şi ele incrementează partea reală a lui *z*.

În principiu, apelul unei funcţii care returnează o referinţă trebuie considerat ca şi cum ar reprezenta un *nume*.

De exemplu, *retreal(z)* este echivalent cu numele calificat *z.x*.

Fie tipul:

```
typedef struct {
 COMPLEX z;
 double modul;
 double arg;
} NRC;
```

şi declaraţia:

```
NRC nrc;
```

Pentru a ne referi la componentele *x* şi *y* ale datei *nrc* se utilizează

numele calificat:

`nrc.z.x` pentru partea reală

şi

`nrc.z.y` pentru partea imaginară.

De exemplu:

```
nrc.z.x = 3;
nrc.z.y = 4;
```

definesc părţile reală şi respectiv imaginară a componentei z a datei structurate *nrc*.

Acelaşi lucru se realizează folosind următoarele atribuiri:

```
nrc.retreal(nrc.z) = 3;
nrc.retimag(nrc.z) = 4;
```

sau dacă considerăm declaraţia:

```
COMPLEX& rc = nrc.z;
```

atunci aceleaşi atribuiri pot fi scrise astfel:

```
nrc.retreal(rc) = 3;
nrc.retimag(rc) = 4;
```

Exerciţii:

20.13 Să se scrie o funcţie care calculează modulul unui număr complex şi returnează o referinţă la el.

Această funcţie este analogă cu cea definită în exerciţiul 10.1.

FUNCŢIA BXX13

```
double& rmodul(COMPLEX& z)

/* calculeaza modulul numarului complex si returneaza o
   referinta la el */
{
 static double c;

 return c=sqrt(z.x*z.x+z.y*z.y);
}
```

20.14 Să se scrie o funcţie care calculează argumentul unui număr complex şi returnează o referinţă la el.

Această funcţie este analogă cu cea definită în exerciţiul 10.2.

FUNCŢIA BXX14

```
double& rarg(COMPLEX& z)

/* calculeaza argumentul numarului complex si returneaza
   o referinta la el */
{
 static double a;

 if(z.x == 0 && z.y == 0)
     return a=0.0;
 if(z.y == 0)
     if(z.x >0)
         return a=0.0;
     else    //y=0 si x < 0
         return a=PI;
 if(z.x == 0)
     if(z.y > 0)
         return a=PI/2;
     else
         return a=(3*PI)/2;

 //x != 0 si y != 0
 a=atan(z.y/z.x);
 if(z.x < 0)   //x < 0 si y != 0
     return a=a+PI;
 else
     if(z.y < 0)   //x > 0 si y < 0
         return a=2*PI+a;
  //x > 0 si y > 0
 return a;
}
```

Observaţie:

Variabila *a* a fost declarată statică din cauză că funcţia returnează o referinţă la ea (se utilizează instrucţiunea *return a;*).

Dacă variabila *a* ar fi fost automatică, ea s-ar fi alocat pe stivă. În acest caz o referinţă la ea nu are sens, deoarece la revenirea din funcţie se face curăţirea stivei şi deci variabila *a* este dezalocată.

20.15 Să se scrie un program care citeşte numere complexe şi le afişează împreună cu modulul şi argumentul fiecăruia.

PROGRAMUL BXX15

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159265358979

typedef struct {
 double x;
 double y;
}COMPLEX;

#include "bxx13.cpp"  // rmodul
#include "bxx14.cpp"  // rarg

main()
/* citeste numere complexe si le afiseaza impreuna cu
   modulul si argumentul fiecaruia */
{
 COMPLEX complex;

 while(scanf("%lf %lf",&complex.x,&complex.y)==2){
     printf("a+ib=%g+i*(%g)\n",complex.x,
             complex.y);
     printf("modul=%g\targ=%g\n",
           rmodul(complex),rarg(complex));
 }
}
```

20.8. Modificatori

Modificatorii sînt cuvinte cheie care se utilizează în declaraţii sau definiţii de variabile şi funcţii.

Cei mai folosiţi modificatori utilizaţi la definirea funcţiilor sînt:

cdecl, pascal, near, far şi **huge**.

Sensul acestor modificatori este amintit în paragraful 20.2.

În acest paragraf amintim modificatorii utilizaţi mai frecvent în legătură cu variabilele.

Aceştia sînt:

const, near, far şi **huge**.

20.8.1. Modificatorul const

În principiu, modificatorul *const* se utilizează în definiţii sau declaraţii pentru a defini date constante.

Cuvîntul cheie *const* trebuie considerat că modifică tipul unei date restrîngînd modul de utilizare al datei respective.

De obicei, cuvîntul *const* precede, în declaraţie, tipul datei.

Exemple:

1. `const int versiune = 3;`
2. `const double pi = 3.14159265358979;`
3. `const int xcif[] = {'a','b','c','d','e','f'};`

Datele declarate (definite) în acest fel nu pot fi modificate direct, deci atribuiri de forma:

```
versiune = 4;
pi = 3.14159;
xcif[0] = 'A';
```

sînt eronate.

Într-o declaraţie (definiţie) în care se utilizează modificatorul *const*, se poate omite tipul. În acest caz se subînţelege tipul *int*. Astfel, constanta declarată ca mai jos:

```
const an = 1995;
```

are tipul *int*.

Modificatorul *const* are utilizări importante mai ales în legătură cu pointerii.

O construcţie de forma:

const *tip* **nume* = ... ;

defineşte un pointer spre o ***dată constantă***.

De exemplu, fie declaraţia:

```
const char *sir = "abc";
```

Prin această declaraţie *sir* devine un pointer spre un şir constant. Deci atribuirile de forma:

```
*sir = 'A';
*(sir+1) = 'B' etc.
```

sînt eronate.

O construcţie de forma:

tip ***const** *nume* = ... ;

defineşte un *pointer constant* spre o dată care nu este constantă.

De exemplu, fie declaraţiile:

```
char *const psir = "abc";
char *s;
const char *sir = "sir";
```

Atunci, o atribuire de forma:

```
sir = s
```

este corectă, dar:

```
psir = s
```

nu este acceptată deoarece *psir* este un pointer constant. În schimb, atribuirile:

```
*psir = 'A'
*(psir+1) = 'B'
*(psir+2) = 'C'
```

sînt corecte.

Este posibil să se definească un pointer constant spre o dată constantă. În acest scop se utilizează o declaraţie (definiţie) de forma:

const *tip* ***const** *nume* = ... ;

De exemplu, fie declaraţia:

```
const char *const atribut = "IBM PC";
```

În acest caz, *atribut* este un pointer constant spre un şir constant de caractere.

Amintim că modificatorul *const* se utilizează frecvent la declararea parametrilor formali de tip pointer pentru a interzice funcţiilor respective modificarea datelor spre care pointează parametrii respectivi.

Astfel de parametri se întîlnesc, de exemplu, la funcţiile de prelucrare a şirurilor de caractere:

```
unsigned strlen(const char *s);
char *strcpy(char *dest, const char *sursa);
```

etc.

Funcţia *strlen* nu poate modifica data spre care pointează s. În mod analog, funcţia *strcpy* nu poate modifica data spre care pointează *sursa*.

Modificatorul *const* poate fi utilizat şi pentru a proteja o dată returnată de o funcţie.

De exemplu, funcţia de mai jos returnează un pointer spre un şir constant, care nu poate fi modificat în mod accidental (direct).

```
const char *denlun(int i)
/* returneaza un pointer la denumirea lunii a i-a */
{
 static const char *tdl[] = {"luna ilegala",
                             "ianuarie",
                             ...
                             "decembrie"};
 return(i < 1 || i > 12) ? tdl[0] : tdl[i];
}
```

Protecţia datelor obţinută cu ajutorul modificatorilor *const* nu este totală.

Într-adevăr, dacă vom considera declaraţiile de mai jos:

```
const char *sir = "abc";
char *s;
```

atunci o atribuire de forma:

```
s = sir
```

este eronată deoarece *sir* şi *s* sînt pointeri de tipuri diferite:

sir este un pointer spre o dată *constantă* de tip caracter, iar *s* este un pointer spre o dată de tip caracter.

În schimb, o atribuire de forma:

```
s = (char *)sir
```

este corectă. În continuare s pointează spre aceeaşi dată ca şi *sir*.

Cum *s* nu a fost declarat cu ajutorul modificatorului *const*, rezultă că atribuirile de mai jos sînt legale:

```
*s = 'A';
*(s+1) = 'B';
*(s+2) = 'C';
```

În felul acesta, data "abc", protejată cu ajutorul modificatorului *const*, a putut fi modificată cu ajutorul pointerului s. O modificare de acest fel se

spune că este o *modificare indirectă.*

Modificatorul *const* nu permite *modificarea directă* a datei, adică modificări de forma:

```
*sir = ...
*(sir+1) = ...
*(sir+2) = ...
```

În felul acesta, modificatorul *const* protejează datele împotriva unor modificări directe care pot surveni în urma unor erori.

20.8.2. Modificatorii near, far şi huge

Aceşti modificatori sînt specifici limbajului TURBO C++. Ei dau posibilitatea utilizatorului de a defini explicit dimensiunea pointerilor: 16 sau 32 biţi.

În cazul limbajului C, modelul memoriei (tiny, small, ...) determină formatul intern al pointerilor.

În C++ se pot folosi declaraţii de forma:

*tip modificator *nume;*

unde:

modificator - Este unul din cuvintele cheie: *near*, *far* sau *huge*.

Modificatorul *near* specifică faptul că pointerul se reprezintă pe 16 biţi. Ceilalţi modificatori forţează reprezentarea pointerilor pe 32 de biţi.

Utilizarea acestor modificatori poate conduce, uneori, la economie de memorie şi de timp calculator evitînd lucrul cu pointerii pe 32 de biţi cînd acesta nu este necesar.

Diferenţa dintre modificatorii *far* şi *huge* se manifestă la calculele cu operanzi de tip pointer. Operaţiile cu pointeri de tip *far* pot conduce uneori la rezultate eronate spre deosebire de cele realizate asupra pointerilor de tip *huge*, operaţii care se efectuează totdeauna corect, dar sînt mai costisitoare în timp.

Aceşti modificatori nu sînt necesari pentru programe de dimensiuni relativ mici şi care nu utilizează volume mari de date.

Programele din volumul de faţă se încadrează în această categorie şi de aceea nu vom utiliza aceşti modificatori în exerciţiile care urmează.

O excepţie de la această regulă o constituie funcţiile standard pentru gestiunea grafică a ecranului care implică modificatorul *far*.

20.9. Cuvîntul cheie void

Cuvîntul cheie **void** are diferite utilizări în funcţie de context. O primă utilizare a lui *void* este în legătură cu funcţiile care nu returnează o valoare la revenirea din ele:

void *nume*(...)

Tot în antetul unei funcţii se poate utiliza cuvîntul *void* între parantezele care urmează după numele funcţiei. În acest caz, *void* indică faptul că funcţia nu are parametri.

Utilizarea lui *void* pentru a indica absenţa parametrilor nu este obligatorie. În schimb, este obligatoriu ca *void* să apară înaintea numelui funcţiei cînd aceasta nu returnează o valoare. Aceleaşi reguli se aplică şi la prototipul funcţiilor.

Exemple:

1. `void f(void);`

   Funcţia *f* nu are parametri şi nu returnează o valoare.

2. `void g();`

   Funcţia *g* nu are parametri şi nu returnează o valoare.

O altă utilizare a cuvîntului cheie *void* este aceea de a defini un tip pointer universal:

void *

Un pointer spre *void* nu are tipul precizat.

Un pointer spre un tip precizat se poate atribui direct unui pointer spre *void*.

Fie declaraţiile:

tip **p*;
void **pv*;

unde:

tip - Este diferit de *void*.

În acest caz, o atribuire de forma:

```
pv = p;
```

este corectă.

În schimb, atribuirea inversă, adică:

```
p = pv;
```

este eronată.

Astfel de atribuiri se pot realiza numai folosind conversii explicite, ca mai jos:

```
p = (tip *)pv.
```

Nu se pot face operaţii cu pointeri spre *void*. Deci, expresiile de forma:

```
pv++
pv--
pv+n
pv-n
```

sînt eronate.

Pentru a putea face operaţii cu astfel de pointeri este necesar să se folosească conversii explicite:

```
(int *)pv +n
(double *)pv-n
etc.
```

În general, se vor evita astfel de operaţii deoarece ele pot conduce la erori.

20.10. Structuri

În limbajul C++ se pot defini tipuri noi folosind declaraţia *struct* ca în limbajul C(vezi capitolul 10).

De asemenea, se pot atribui nume la tipuri predefinite sau definite de utilizator folosind construcţia *typedef*.

Pentru tipurile definite cu ajutorul lui *struct* putem defini date ca mai jos:

struct *nume* {
  ...
};
struct *nume nume1,nume2,...*;

În limbajul C++ nu mai este necesară utilizarea lui *struct* la declararea structurilor *nume1*, *nume2*, ... , de tipul *nume*:

Deci, vom scrie mai simplu:

nume nume1, *nume2*, ... ;

unde:

nume - Este tipul utilizator definit printr-o construcţie *struct* ca mai sus.

În felul acesta, în limbajul C++, nu mai este necesară denumirea tipurilor structurate (definite cu ajutorul lui *struct*) prin intermediul construcţiei *typedef*.

În limbajul C++ o structură poate fi transferată direct prin parametri. Astfel, dacă avem declaraţia:

struct *nume* {
...
};

atunci datele de tip *nume* pot fi transferate prin parametri în 3 moduri:

- direct : tip f1(nume par) { ... }
- pointer spre structură : tip f2(nume *par) { ... }
- referinţă la structură : tip f3(nume& par) { ... }

De asemenea, o funcţie poate returna o structură, un pointer spre structură sau o referinţă la o structură:

nume g1(...) { ... } - returnează o dată de tip nume;
nume *g2(...) { ... } - returnează un pointer spre o dată de tip nume;
nume&g3(...) { ... } - returnează o referinţă la o dată de tip nume.

Structurile de acelaşi tip se pot atribui direct una alteia.

Fie:

```
nume a1,a2;
```

Atunci atribuirea:

```
a1 = a2
```

este legală.

20.11. Reuniune

În limbajul C++, faţă de limbajul C, reuniunile pot fi iniţializate. Iniţializarea se poate realiza numai pentru prima componentă a ei.

Exemplu:

```
union {
 double pi;
```

```
 int i;
 char c;
} u = {3.14159265358979};
```

Cuvîntul *union* poate fi omis la declararea datelor de tip reuniune.

Astfel, fie declaraţia:

```
union alternative {
 int i;
 float f;
 double d;
 char c;
};
```

În continuare se pot declara, defini, date de tip *alternative* ca mai jos:

```
alternative zonal,zona2;
```

O altă facilitate existentă în C++ este posibilitatea de a utiliza reuniuni *anonime* ca şi componente într-o structură. Aceasta permite accesul mai simplu la componentele structurii.

Exemplu:

```
struct {
 int tip;
 union {
     int i;
     float f;
     double d;
 };
}a;
```

Reuniunea, componentă a structurii *a*, este anonimă. La componentele ei ne referim ca mai jos:

a.i = 1234;

...

a.f = 3.14159;

...

a.d = 3.14159265;

20.12. Tipul enumerare

În limbajul C++, tipul enumerare se defineşte ca în limbajul C. Cu toate acestea există unele diferenţe.

Diferența esențială constă în aceea că în limbajul C++ se fac teste asupra valorilor care se atribuie unei date de tip enumerare. De aceea, la o dată de tip enumerare se pot atribui numai enumeratorii (componentele) ei.

Exemplu:

Fie declarațiile:

```
enum sapt {luni=1, marti, miercuri, joi, vineri,
           simbata, duminica};
enum sapt ziua;
```

Expresiile de mai jos sînt legale:

```
ziua = vineri;
if(ziua >= luni && ziua <= vineri)
// zi lucratoare
else
// zi de odihna
```

În C++, o expresie de forma:

```
ziua = 2
```

este eronată, deși conform declarației de mai sus, *marti* are valoarea 2.

Enumeratorii sînt de tip *int*.

Ca și în cazul structurilor, cuvîntul *enum* poate fi omis la declararea datelor de tip enumerare.

Astfel, *ziua* se poate declara mai simplu scriind:

```
sapt ziua;
```

20.13. Supraîncărcarea funcțiilor

În limbajul C, funcțiile utilizate într-un program au nume *distincte*.

În limbajul C++ există posibilitatea ca funcții diferite să aibă un *același* nume.

De obicei, funcțiile "înrudite" pot fi denumite printr-un același nume.

Cazurile cele mai frecvente de funcții cu același nume sînt funcțiile care realizează același proces de calcul dar asupra unor date de tipuri diferite. De exemplu, în limbajul C există funcții, în biblioteca sistemului, cu nume distincte pentru calculul valorii absolute:

abs - Pentru calculul valorii absolute a unei date de tip *int*.

labs - Pentru calculul valorii absolute a unei date de tip *long*.
fabs - Pentru calculul valorii absolute a unei date de tip *double*.

Aceste funcţii au următoarele prototipuri:

```
int abs(int);
long labs(long);
double fabs(double);
```

Primele două prototipuri se află în fişierul *stdlib.h*, iar cel de al treilea în fişierul *math.h*.

În limbajul C++, aceste funcţii pot fi numite cu un acelaşi nume.

În acest caz, ele se disting prin tipul parametrului. Astfel, în limbajul C++ este posibil să definim aceste 3 funcţii ca mai jos:

1.
```
int abs(int n)
{
  return n < 0 ? -n : n;
}
```

2.
```
long abs(long n)
{
  return n < 0l ? -n : n;
}
```

3.
```
double abs(double n)
{
  return n < 0.0 ? -n : n;
}
```

La un apel de forma:

x = abs(*expresie*);

compilatorul C++ apelează una din cele 3 funcţii *abs* definite mai sus, ţinînd seama de tipul parametrului efectiv, adică de *expresie* dintre paranteze. Astfel, dacă *expresie* are tipul *int*, atunci se apelează funcţia definită la punctul 1, dacă *expresie* are tipul *long*, atunci se apelează funcţia definită la punctul 2, iar dacă *expresie* are tipul *double*, atunci se apelează funcţia de la punctul 3. În felul acesta, numele *abs* reprezintă 3 funcţii pe care compilatorul le poate distinge, la apel, după tipul parametrului efectiv.

Se obişnuieşte să se spună că cele 3 funcţii supraîncarcă numele *abs*.

De asemenea, se spune că numele *abs* este *supraîncărcat* (*overload*).

În general, un nume se spune că este *supraîncărcat* dacă el este un nume comun pentru mai multe funcţii.

Despre funcţiile cu acelaşi nume vom spune că sînt supraîncărcate (**Functions overloading**).

La ora actuală există încetăţenit în limba română şi alţi termeni pentru supraîncărcare, ca de exemplu, *suprapunere*, *redefinire* sau *supradefinire*.

În această carte adoptăm termenul de supraîncărcare.

O condiţie esenţială pentru supraîncărcarea funcţiilor este ca ele să aibă prototipuri *diferite*. În caz contrar, compilatorul nu poate face distincţie, între ele, la apelul lor.

În principiu, numărul şi tipul parametrilor efectivi furnizează un criteriu de selecţie al funcţiilor supraîncărcate.

Avînd în vedere acest fapt, rezultă că la apelul funcţiilor supraîncărcate, de obicei, nu se mai aplică regula simplă de conversii a tipurilor parametrilor efectivi spre tipurile parametrilor formali corespunzători, regulă utilizată în limbajul C.

În acest caz, se pot aplica conversii dacă nu există coincidenţă între tipurile parametrilor efectivi şi cei formali cu care se corespund.

În limbajul C++ se încearcă selectarea acelei funcţii pentru care există coincidenţă între numărul şi tipul parametrilor efectivi (de la apel) şi al parametrilor formali ai funcţiei supraîncărcate. În cazul în care nu se poate stabili o astfel de coincidenţă, se fac şi în acest caz conversii ale parametrilor efectivi. Aceste conversii pot să nu fie unice şi de aceea sînt necesare reguli suplimentare.

De exemplu, reluînd apelul de mai sus:

x = **abs**(*expresie*);

unde *expresie* are tipul *float*, se observă că nu se realizează coincidenţă cu tipul parametrului formal al nici uneia din funcţiile care supraîncarcă numele *abs*. De aceea, în acest caz, se va face o conversie a valorii expresiei de la apel. Sînt posibile 3 conversii:

```
float -> int
float -> long
float -> double
```

În principiu, se aleg conversii care să nu conducă la pierderea de informaţie.

De aceea,în cazul de faţă, sînt excluse primele două conversii.

La un apel, selecţia funcţiei se realizează în mai mulţi paşi şi în ordinea de mai jos:

1. Se caută o corespondenţă exactă şi se apelează funcţia respectivă dacă ea există.
2. Dacă nu se poate determina o funcţie cu coincidenţă exactă, atunci se caută una pe baza efectuării de conversii predefinite (pentru tipuri predefinite) care nu conduc la pierderi de informaţie, adică se fac conversii fără trunchieri.
3. Dacă nu se poate determina o funcţie conform punctelor precedente, atunci se încearcă selectarea unei funcţii folosind conversii pentru tipuri predefinite care pot să conducă la trunchieri (de exemplu, date de tip *double* se convertesc în date de tip *float*, *long* sau *int*).
4. Dacă în paşii precedenţi nu se poate determina funcţia de apelat, atunci se caută o funcţie admiţînd conversii nu numai pentru tipuri predefinite ci şi pentru cele definite de utilizator.

În cazul în care nu se poate selecta o funcţie *unică* parcurgînd în ordine paşii de mai sus, compilatorul C++ semnalează eroare.

Mai tîrziu, se va reveni asupra problemei apelurilor funcţiilor în legătură cu conversia datelor de tip utilizator.

Exerciţii:

20.16 Să se supraîncarce numele *modul* cu funcţii de calcul a valorii absolute pentru numere întregi şi neîntregi, iar pentru numere complexe se va calcula modulul.

FIŞIERUL BXX16

```
int modul(int i)
// returneaza valoarea absoluta a lui i
{
 return i < 0 ? -i : i;
}

long modul(long n)
// returneaza valoarea absoluta a lui n
{
 return n < 0 ? -n : n;
}
```

```
double modul(double d)

// returneaza valoarea absoluta a lui d
{
 return d < 0 ? -d : d;
}

typedef struct {
 double x;
 double y;
} COMPLEX;

double modul(COMPLEX z)

// returneaza modulul lui z
{
 return sqrt(z.x * z.x+z.y * z.y);
}

long double modul(long double ld)

// returneaza valoarea absoluta a lui ld
{
 return ld < 0 ? -ld : ld;
}
```

20.17 Să se definească funcţii supraîncărcate cu numele *afiseaza*, care să afişeze numere şi şiruri de caractere.

Funcţiile au unul sau mai mulţi parametri. Primul parametru defineşte numărul de afişat. Parametrul al doilea, dacă este prezent, defineşte dimensiunea minimă a cîmpului în care se afişează data. Data se afişează cadrată în dreapta şi eventual este precedată de spaţii.

Parametrul al treilea, dacă este prezent, indică numărul de zecimale în cazul numerelor neîntregi sau numărul de caractere în cazul în care se afişează şiruri de caractere.

FIŞIERUL BXX17

```
char *format(const char *s1,int cimp);
char *format(const char *s1,const char *s2);
```

```
void afiseaza(int i,int cimp=5)
/* afiseaza valoarea lui i in cimpul de dimensiune minima
  egala cu cimp */
{
 char *sir,*sirl;

 sirl = format("%",cimp);
 sir = format(sirl,"d");
 printf(sir,i);
 delete sir;
 delete sirl;
}

void afiseaza (long l,int cimp = 10)

/* afiseaza valoarea lui l in cimpul de dimensiune minima egala
  cu cimp */
{
 char *sir,*sirl;

 sirl = format("%",cimp);
 sir = format(sirl,"ld");
 printf(sir,l);
 delete sir;
 delete sirl;
}

void afiseaza(double d,int cimp = 15,
              int precizie = 6)

/* afiseaza valoarea lui d in cimpul de dimensiune minima
   egala cu cimp si cu precizie definita de precizie */
{
 char *sir,*sirl,*sir2,*sir3;

 sirl = format("%",cimp);
 sir2 = format(sirl,".");
 sir3 = format(sir2,precizie);
 sir = format(sir3,"g");
 printf(sir,d);
 delete sir;
 delete sirl;
 delete sir2;
```

```
 delete sir3;
}

void afiseaza(long double lf,
              int cimp = 15,int precizie = 6)
```

/* afiseaza valoarea lui lf in cimpul de dimensiune minima egala cu cimp si cu precizia definita de precizie */

```
{
 char *sir,*sir1,*sir2,*sir3;

 sir1 = format("%",cimp);
 sir2 = format(sir1,".");
 sir3 = format(sir2,precizie);
 sir = format(sir3,"Lg");
 printf(sir,lf);
 delete sir;
 delete sir1;
 delete sir2;
 delete sir3;
}

void afiseaza(COMPLEX z,int cimp = 15,
              int precizie = 6)
```

/* afiseaza numarul complex z in cimpul cimp si cu precizia definita de precizie */

```
{
 char *sir;
 char *sir1,*sir2;

 sir  = format("%",cimp);
 sir1  = format(sir,".");
 delete sir;
 sir  = format(sir1,precizie);
 delete sir1;
 sir2  = format(sir,"g ");
 delete sir;
 sir1 = format("Partea reala = ",sir2);
 sir = format(sir1,"Partea imaginara = ");
 delete sir1;
 sir1 = format(sir,sir2);
 printf(sir1,z.x,z.y);
```

```
 delete sir;
 delete sir1;
 delete sir2;
}

void afiseaza(const char *s,int cimp = 70,
              int precizie = 70)
```

/* afiseaza sirul de caractere spre care pointeaza s intr-un cimp de lungime minima egala cu valoarea lui cimp si cu precizia egala cu precizie */

```
{
 char *sir,*sir1,*sir2,*sir3;

 sir1 = format("%",cimp);
 sir2 = format(sir1,".");
 sir3 = format(sir2,precizie);
 sir = format(sir3,"s");
 printf(sir,s);
 delete sir;
 delete sir1;
 delete sir2;
 delete sir3;
}

char *format(const char *s,int n)
```

/* - construieste un sir prin concatenarea caracterelor spre care pointeaza s cu cele rezultate prin conversia lui n in zecimal;
 - sirul rezultat se pastreaza in memoria heap. */

```
{
 char zona[10];

 itoa(n,zona,10);
 return format(s,zona);
}

char *format(const char *s1,const char *s2)
```

/* construieste un sir prin concatenarea sirului spre care pointeaza s2 la sfirsitul sirului spre care pointeaza s1 */

```
{
 char *mem;
```

```
 mem = new char[strlen(s1)+strlen(s2)+1];
 strcpy(mem,s1);
 strcat(mem,s2);
 return mem;
}
```

20.18 Să se scrie un program care afişează modulele numerelor citite de la intrarea standard.

PROGRAMUL BXX18

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include <math.h>
#include "BXX16.CPP"  // functiile supraincarcate modul
#include "BXX17.CPP"  // functiile supraincarcate afiseaza

main()
/* afiseaza modulele numerelor citite de la intrarea standard */

{
 int i;
 char c;
 char cont[] ="\n Actionati o tasta pentru\
   a continua\n";
 char intreg_int[] = "\nint=";
 char t[255];

 for(;;) {
    for(;;) {
       afiseaza(intreg_int,strlen(intreg_int));
       if(gets(t)==0)
          exit(1);
       if(sscanf(t,"%d",&i)==1)
          break;
       afiseaza("nu s-a tastat un intreg",10);
       afiseaza("se reia citirea intregilor\n");
    }
    afiseaza("\n\n\t\t tip int\n\n",10);
    afiseaza(modul(i));
    afiseaza(cont,50);
    getch();
```

```
      afiseaza("Alt int? Se raspunde cu D sau N ");
      if((c = getch()) != 'D' && c != 'd')
         break;
   } // sfirsit for pentru int

   char intreg_long[] = "\nlong=";
   long l;

   for(;;) {
      for(;;) {
         afiseaza(intreg_long,10);
         if(gets(t) == 0)
            exit(1);
         if(sscanf(t,"%ld",&l) == 1)
            break;
         afiseaza("nu s-a tastat un intreg\n",10);
         afiseaza("se reia citirea intregilor\n");
      }
      afiseaza("\n\n\t tip long\n\n",10);
      afiseaza(modul(l));
      afiseaza(cont,50);
      getch();
      afiseaza("Alt long ? Se raspunde\
               cu D sau N ");
      if((c = getch()) != 'D' &&c != 'd')
         break;
   } // sfirsit for pentru long

   char flotant_simplu[] = "\nflotant simpla\
                           precizie=";
   char rel[] = "se reia citirea numarului\n";
   float f;
   char er[] = "nu s-a tastat un numar\n";

   for(;;) {
      for(;;) {
         afiseaza(flotant_simplu,10);
         if(gets(t) == 0)
            exit(1);
         if(sscanf(t,"%f",&f) == 1)
            break;
         afiseaza(er,10);
         afiseaza(rel,10);
```

```
    }
    afiseaza("\n\n\t tip float \n\n",10);
    afiseaza(modul(f));
    afiseaza(cont,50);
    getch();
    afiseaza("Alt float ? Se raspunde cu\
             D sau N ");
    if((c=getch()) != 'D' && c != 'd')
         break;
} //sfirsit for pentru float

char flotant_dublu[] = "\nflotant\
        dubla precizie=";
double d;

for(;;) {
   for(;;){
       afiseaza(flotant_dublu,10);
       if(gets(t) == 0)
          exit(1);
       if(sscanf(t,"%lf",&d)==1)
          break;
       afiseaza(er,10);
       afiseaza(rel,10);
   }
   afiseaza("\n\n\t tip double\n\n",10);
   afiseaza(modul(d),25,16);
   afiseaza(cont,50);
   getch();
   afiseaza("Alt double ? Se raspunde\
            cu D sau N ");
   if((c=getch()) != 'D' && c != 'd')
      break;
} //sfirsit for pentru double

char flotant_lung[] = "\nflotant lung=";
long double ld;

for(;;) {
   for(;;) {
       afiseaza(flotant_lung,10);
       if(gets(t) == 0)
          exit(1);
```

```
            if(sscanf(t,"%Lf",&ld) == 1)
              break;
            afiseaza(er,10);
            afiseaza(rel,10);
          }
          afiseaza("\n\n\t tip long double\n\n",10);
          afiseaza(modul(ld),30,20);
          afiseaza(cont,50);
          getch();
          afiseaza("Alt long double? Se raspunde\
                    cu D sau N ");
          if((c=getch()) != 'D' && c != 'd')
            break;
        } //sfirsit for pentru long double

        char n_complex[] = "\ncomplex Partea reala=";
        COMPLEX complex;
        for(;;) {
          for(;;) {
            afiseaza(n_complex,10);
            if(gets(t) == 0)
              exit(1);
            if(sscanf(t,"%lf",&complex.x) != 1) {
              afiseaza (er,10);
              afiseaza(rel,10);
              continue;
            }
            afiseaza("\nPartea imaginara=",10);
            if(gets(t) == 0)
              exit(1);
            if(sscanf(t,"%lf",&complex.y) == 1)
              break;
            afiseaza(er,10);
            afiseaza(rel,10);
          }
          afiseaza("\n\n\t tip complex\n\n",10);
          afiseaza(modul(complex),25,16);
          afiseaza(cont,50);
          getch();
          afiseaza("Alt complex ? Se raspunde cu\
                    D sau N ");
          if((c = getch()) != 'D' && c != 'd')
```

```
        break;
    } //sfirsit for pentru complex
}
```

Observaţii:

1. Funcţiile supraîncărcate utilizate în acest program sînt:
   - *afiseaza*;
   - *format*;
   - *modul*.

   Ele se selectează prin coincidenţă între tipurile parametrilor efectivi şi formali care se corespund prin poziţie.

2. Un apel de forma:

   ```
   afiseaza(sir);
   ```

   unde *sir* este un pointer spre un şir de caractere, afişează şirul respectiv într-un cîmp de minimum 70 de caractere. Aceasta deoarece parametrul implicit *cimp* al funcţiei *afiseaza* are valoarea 70. În cazul în care şirul care se afişează are mai puţine caractere, el se afişează cadrat în dreapta în cîmpul respectiv.

   Dacă şirul conţine mai mult de 70 de caractere, atunci numai primele 70 se vor afişa, deoarece parametrul implicit *precizie* are valoarea 70.

   Un apel de forma:

   ```
   afiseaza(sir,10);
   ```

   atribuie parametrului *cimp* valoarea 10. În acest caz, şirurile de caractere care conţin mai puţin de 10 caractere se afişează într-un cîmp de 10 caractere şi cadrat în dreapta.

Şirurile cu peste 10 caractere, dar care nu depăşesc 70, se afişează pe atîtea caractere cîte intră în compunerea lor.

Parametrul implicit *precizie*, nefiind definit în apelul de mai sus, şirurile cu peste 70 de caractere se afişează trunchiat şi anume numai primele 70 de caractere vor fi afişate.

20.19 Să se scrie funcţii care supraîncarcă numele *putere* şi care ridică la putere numere de diferite tipuri.

FIŞIERUL BXX19

```
int putere(int a,int b)
```

```
// ridica pe a la puterea b
{
 int c;
 int i;

 if(a == 0)
     return 0;
 if(b < 0)
     return 0;
 for(i = 1,c = 1;i <= b;i++)
     c *= a;
     return c;
 }

long putere(long a,int b)

// ridica pe a la puterea b
{
 long c;
 int i;

 if(a == 0)
     return 0;
 if(b < 0)
     return 0;
 for(i = 1,c = 1;i <= b;i++)
     c *= a;
     return c;
 }

double putere(double a,int b)

// ridica pe a la puterea b
{
 double c;
 int i,j;

 if(a == 0)
     return 0.0;
 j = b < 0 ? -b : b;     //j = abs(b)
 for(i = 1,c = 1.0; i <= j;i++)
     c *= a;
     return b < 0 ? 1.0/c : c;
```

```
}

double putere(double a,double b)

// ridica pe a la puterea b
{
 return pow(a,b);
}
```

20.20 Să se scrie un program care citeşte şiruri de perechi de numere de forma (x,y) şi afişează, pentru fiecare pereche, valoarea lui *x* la puterea *y*. Numerele *x* şi *y* sînt de diferite tipuri.

PROGRAMUL BXX20

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include "BXX19.CPP"

main()
/* citeste perechi de numere de forma (a,b) si afiseaza pe a la b */
{
 char t[255];
 char baza[] = "baza\n";
 char exp[] = "exponent\n";
 char erint[] = "nu s-a tastat un intreg\n";
 int i,j;
 int c;

 for(;;) {
     for(;;) {
         printf(baza);
         printf("int=");
         if(gets(t) == 0)
             exit(1);
         if(sscanf(t,"%d",&i) == 1)
             break;
         printf(erint);
     }
     for(;;) {
         printf(exp);
         printf("int=");
```

```
        if(gets(t) == 0)
           exit(1);
        if(sscanf(t,"%d",&j) == 1)
           break;
        printf(erint);
     }
     int k = putere(i,j);
     printf("i = %d\t j = %d\t i**j = %d\n",i,j,k);
     printf("Alt int**int ? Se raspunde\
          cu D sau N \n");
     if((c = getch()) != 'D' && c != 'd')
        break;
  } // Sfirsit int**int

  long u;

  for(;;) {
     for(;;) {
        printf(baza);
        printf("long=");
        if(gets(t) == 0)
           exit(1);
        if(sscanf(t,"%ld",&u) == 1)
           break;
        printf("nu s-a tastat un intreg de tip\
             long\n");
     }
     for(;;) {
        printf(exp);
        printf("int=");
        if(gets(t) == 0)
           exit(1);
        if(sscanf(t,"%d",&j) == 1)
           break;
        printf(erint);
     }
     long m = putere(u,j);
     printf("u = %ld\t j = %d\t u**j = %ld\n",
          u,j,m);
     printf("Alt long**int ? Se raspunde\
          cu D sau N \n");
     if((c = getch()) != 'D' && c != 'd')
        break;
```

```
} // Sfirsit long**int

float f;
char ern[] = "Nu s-a tastat un numar\n";

for(;;) {
   for(;;) {
      printf(baza);
      printf("float=");
      if(gets(t) == 0)
         exit(1);
      if(sscanf(t,"%f",&f) == 1)
         break;
      printf(ern);
   }
   for(;;) {
      printf(exp);
      printf("int=");
      if(gets(t) == 0)
         exit(1);
      if(sscanf(t,"%d",&j) == 1)
         break;
      printf(erint);
   }
   float g = putere(f,j);
   printf("f = %g\t j = %d\t f**j = %g\n",f,j,g);
   printf("Alt float**int ? Se raspunde\
           cu D sau N \n");
   if((c = getch()) != 'D' && c != 'd')
      break;
} // Sfirsit float**int

double d;

for(;;) {
   for(;;) {
      printf(baza);
      printf("double=");
      if(gets(t) == 0)
         exit(1);
      if(sscanf(t,"%lf",&d) == 1)
         break;
      printf(ern);
```

```
    }
    for(;;) {
        printf(exp);
        printf("int=");
        if(gets(t) == 0)
            exit(1);
        if(sscanf(t,"%d",&j) == 1)
            break;
        printf(erint);
    }
    double e = putere(d,j);
    printf("d = %g\t j = %d\t d**j = %g\n",d,j,e);
    printf("Alt double**int ? Se raspunde cu\
            D sau N \n");
    if((c = getch()) != 'D' && c != 'd')
        break;
} // Sfirsit double**int

double h;

for(;;) {
    for(;;) {
        printf(baza);
        printf("double=");
        if(gets(t) == 0)
            exit(1);
        if(sscanf(t,"%lf",&d) == 1)
            break;
        printf(ern);
    }
    for(;;) {
        printf(exp);
        printf("double=");
        if(gets(t) == 0)
            exit(1);
        if(sscanf(t,"%lf",&h) == 1)
            break;
        printf(ern);
    }
    double g = putere(d,h);
    printf("d = %g\t h = %g\t d**h = %g\n",d,h,g);
    printf("Alt double**double ? Se raspunde cu\
            D sau N \n");
```

```
        if((c = getch()) != 'D' && c != 'd')
            break;
    } // Sfirsit double**double

    float fexp;
    long double ld;

    for(;;) {
        for(;;) {
            printf(baza);
            printf("long double=");
            if(gets(t) == 0)
                exit(1);
            if(sscanf(t,"%Lf",&ld) == 1)
                break;
            printf(ern);
        }
        for(;;) {
            printf(exp);
            printf("float=");
            if(gets(t) == 0)
                exit(1);
            if(sscanf(t,"%f",&fexp) == 1)
                break;
            printf(ern);
        }
        double p = putere(ld,fexp);
        printf("ld = %Lf\t fexp = %f\t \
              ld**fexp = %g\n",ld,fexp,p);
        printf("Alt long double**float ? Se raspunde\
              cu D sau N \n");
        if((c = getch()) != 'D' && c != 'd')
            break;
    } // Sfirsit long double**float

    float q,r;

    for(;;) {
        for(;;) {
            printf(baza);
            printf("float=");
            if(gets(t) == 0)
                exit(1);
```

```
            if(sscanf(t,"%f",&q) == 1)
               break;
            printf(ern);
         }
         for(;;) {
            printf(exp);
            printf("float=");
            if(gets(t) == 0)
               exit(1);
            if(sscanf(t,"%f",&r) == 1)
               break;
            printf(ern);
         }
         double s = putere(q,r);
         printf("q = %g\t r = %g\t q**r = %g\n",q,r,s);
         printf("Alt float**float ? Se raspunde\
               cu D sau N \n");
         if((c = getch()) != 'D' && c != 'd')
            break;
      } // Sfirsit float**float
   }
```

Observaţii:

1. În acest program se utilizează funcţia *putere* supraîncărcată cu următoarele tipuri predefinite:

	Rezultat returnat	baza	exponentul
1.	int	int	int
2.	long	long	int
3.	double	double	int
4.	double	double	double

2. Funcţia *putere* se apelează cu parametri efectivi de diferite tipuri predefinite, după cum urmează:

   a. *putere*(**int,int**)
   b. *putere*(**long,int**)
   c. *putere*(**float,int**)
   d. *putere*(**double,int**)
   e. *putere*(**double,double**)
   f. *putere*(**long double,float**)

g. *putere*(**float,float**)

Funcţiile *putere* supraîncărcate se selectează astfel:

La punctul	**Se selectează funcţia de la punctul**
a	1: selectare prin coincidenţă;
b	2: selectare prin coincidenţă;
c	3: selectare prin conversie: baza de tip *float* se converteşte spre tipul *double*;
d	3: selectare prin coincidenţă;
e	4: selectare prin coincidenţă;
f	4:selectare prin conversie: baza de tip *long double* se converteşte spre *double* şi exponentul de tip *float* spre *double*;
g	4: selectare prin conversia ambilor parametri din *float* spre *double*.

20.14. Funcţii inline

În paragraful 15.1. s-a arătat posibilitatea de definire a macrourilor în limbajul C. Această facilitate este prezentă şi în limbajul C++, dar utilizarea este diminuată prin prezenţa funcţiilor *inline*.

Amintim că un macro se defineşte printr-o construcţie de forma:

#define *nume(p1,p2, ... ,pn) text*

Parametrii *p1*, *p2*, ..., *pn*, numiţi parametrii formali, apar în textul de substituţie *text*.

Un macro poate fi apelat prin construcţii de forma:

nume(e1,e2, ... ,en)

unde:

e1,e2, ... ,en - Sînt parametrii efectivi ai apelului.

Apelul macroului se înlocuieşte cu *text* după ce, în prealabil, parametrii formali *pi*, i = 1,2,...,n s-au înlocuit, în textul *text*, cu parametrii efectivi corespunzători *ei*, i = 1,2,...,n. Această înlocuire a apelului macroului se numeşte *expandare*.

Expandarea macrourilor diferă de modul în care se realizează apelul funcţiilor obişnuite.

La apelul unei funcţii obişnuite nu se substituie apelul funcţiei prin

corpul ei, ci se realizează un salt la zona de memorie în care se păstrează corpul funcţiei respective.

La terminarea execuţiei funcţiei se revine în punctul imediat următor apelului. Un apel de această formă, numit şi apel cu revenire, implică diferite operaţii suplimentare.

În cazul în care funcţia este foarte simplă (2- 3 instrucţiuni), operaţiile implicate de apel şi revenire pot fi mai costisitoare decît cele implicate de funcţia însăşi. De aceea, în astfel de situaţii, ar fi util ca apelul funcţiilor să se realizeze la fel ca apelul de macro, adică prin expandare (înlocuirea apelului prin corpul funcţiei).

Acest lucru este posibil dacă antetul funcţiei este precedat de cuvîntul cheie *inline*.

O astfel de funcţie se numeşte funcţie *inline*.

Exemplu:

Pentru calculul maximului dintre două numere se poate defini macroul de mai jos:

```
#define MAX(x,y) (x) < (y) ? (y) : (x)
```

Fie secvenţa pentru apelul macroului MAX:

```
int a,b,c;
c = MAX(a+b,a-b);
```

La preprocesare, se substituie apelul macroului prin textul său şi se obţine:

```
c = (a+b) < (a-b) ? (a-b) : (a+b);
```

Calculul maximului se poate realiza folosind funcţia *inline* de mai jos:

```
inline int max(int x,int y)
{
 return x < y ? y : x;
}
```

Instrucţiunea de atribuire de mai sus se scrie:

```
c = max(a+b,a-b);
```

În acest caz, apelul funcţiei *max* se schimbă la compilare prin corpul ei. Se obişnuieşte să se spună că apelul funcţiei se realizează prin expandarea ei. În felul acesta se elimină operaţiile specifice de la apelul şi revenirea din funcţiile obişnuite.

Funcţiile *inline* au unele avantaje faţă de macrouri. Astfel, la expandarea

unui macro nu se ţine seama de tipul parametrilor. De aceea, nu se realizează nici un control asupra tipurilor parametrilor efectivi sau conversii ale valorilor acestora spre tipurile parametrilor formali.

La apelul unui macro, parametrii formali se substituie în textul din corpul macroului prin parametrii efectivi corespunzători.

Această substituţie se realizează înlocuind un şir de caractere (numele parametrului formal) printr-un alt şir de caractere care defineşte parametrul efectiv corespunzător.

De aici rezultă şi un alt dezavantaj al macrourilor şi anume acela al *efectelor secundare*, care în cazul macrourilor sînt mult mai greu de pus în evidenţă decît în cazul funcţiilor. Efectele secundare sau *colaterale* apar la calculul expresiilor care conţin operanzi ale căror valori se schimbă la evaluarea lor. Astfel de efecte se întîlnesc mai ales la utilizarea, în expresii, a operatorilor de incrementare şi decrementare.

De exemplu, atribuirea:

```
c = MAX(a++,b++);
```

se expandează astfel:

```
c = (a++) < (b++) ? (b++) : (a++);
```

Efectul acestei instrucţiuni este diferit de cel al instrucţiunii obţinute apelînd funcţia inline *max* în locul macroului *MAX*:

```
c = max(a++,b++)
```

Acest apel atribuie lui *c* maximul dintre *a* şi *b*, funcţia *max* nefiind influenţată de incrementarea valorilor lui *a* şi *b*.

Valoarea lui *c* în urma apelului macroului *MAX* este influenţată de efectele secundare rezultate din incrementarea valorilor variabilelor *a* şi *b*.

Menţionăm că efectele secundare pot fi prezente şi în cazul funcţiilor, dar în acest caz ele pot fi controlate mai uşor.

De exemplu, apelul funcţiei *max* în instrucţiunea de mai jos:

```
c = max(b+a++,2*a++);
```

atribuie lui *c* o valoare care poate să fie diferită pentru implementări diferite ale limbajului C++. Nu există o regulă pentru ordinea de evaluare a parametrilor efectivi şi din această cauză nu putem şti dacă evaluarea expresiei:

```
2*a++
```

se realizează înainte sau după evaluarea expresiei b+a++.

De aceea, se recomandă evitarea apelurilor de acest tip prin introducerea unor atribuiri prealabile pentru parametri efectivi cu efecte secundare:

```
e1 = b+a++;
e2 = 2*a++;
c = max(e1,e2);
```

Avînd în vedere avantajele funcţiilor *inline* faţă de macrouri, rezultă interesul scăzut în utilizarea macrourilor în limbajul C++.

Construcţia *#define* în limbajul C++ se utilizează frecvent în legătură cu compilarea condiţionată, precum şi în cazul funcţiilor de bibliotecă utilizate atît în programe C cît şi în programe C++.

Funcţiile *inline* se vor utiliza totuşi cu precauţie, deoarece ele sînt avantajoase numai cînd sînt simple. O funcţie *inline* ce conţine mai mult de 3 instrucţiuni poate deveni ineficientă deoarece expandarea ei în program în locul apelurilor sale poate conduce la o creştere substanţială a dimensiunii programului.

Funcţiile *inline* au anumite dezavantaje faţă de funcţiile obişnuite. Astfel, aceste funcţii nu pot fi externe şi deci ele pot fi definite şi utilizate într-un singur modul al programului.

De asemenea, corpul unei funcţii *inline* nu poate conţine secvenţe ciclice (secvenţe introduse prin instrucţiunile *while*, *for* şi *do-while*).

Atributul *inline* este neglijat de către compilator în cazul în care funcţia nu poate fi tratată ca atare.

De exemplu, dacă o funcţie cu atributul *inline* conţine o instrucţiune ciclică, atunci atributul *inline* va fi neglijat şi funcţia respectivă va fi tratată în mod obişnuit. Compilatorul emite un mesaj de avertisment (*warnings*) corespunzător.

Exerciţii:

20.21 Să se scrie un program care citeşte un şir de perechi de numere întregi şi afişează, pentru fiecare pereche, maximul dintre ele.

PROGRAMUL BXX21

```
#include <stdio.h>
inline int max(int x, int y)

/* returneaza maximul dintre x si y */
{
```

```
  return x < y ? y : x;
 }

 main ()
 /* citeste un sir de perechi de intregi si afiseaza, pentru fiecare pereche,
   maximul dintre ele */
 {
  int a,b;

  while(scanf("%d %d", &a,&b) == 2 )
      printf("a=%d\tb=%d\tmax(a,b)=%d\n",a,b,
              max(a,b));
 }
```

20.22 Să se scrie un program care citeşte un şir de perechi de întregi şi afişează, pentru fiecare pereche, maximul dintre valorile lor absolute.

PROGRAMUL BXX22

```
 #include <stdio.h>

 inline int abs(int x)

 /* returneaza valoarea absoluta a lui x */
 {
  return x< 0 ? -x: x;
 }

 inline int maxabs(int x, int y)
 /* returneaza maximul valorilor absolute ale lui x si y */
 {
  int z= abs(x);
  int u= abs(y);

  return z < u ? u : z;
 }

 main()
 /* citeste un sir de perechi de intregi si afiseaza, pentru fiecare pereche,
   maximul dintre valorile lor absolute */
 {
 int a,b;
```

```
while (scanf("%d %d",&a,&b)== 2 )
    printf("a=%d\tb=%d\tmaxabs(a,b)=%d\n",a,b,
            maxabs(a,b));
}
```

20.23 Să se scrie un program care citeşte un şir de perechi de întregi şi afişează, pentru fiecare pereche, maximul dintre valorile lor absolute, apoi realizează acelaşi lucru pentru perechi de numere.

Pentru a termina şirul de perechi de numere întregi se va tasta un caracter oarecare care nu intră în compunerea unui întreg (de exemplu o literă).

PROGRAMUL BXX23

```
#include <stdio.h>

inline int abs(int x)

/* returneaza valoarea absoluta a lui x */
{
 return x< 0 ? -x: x;
}

inline double abs(double x)

/* returneaza valoarea absoluta a lui x */
{
 return x < 0.0 ? -x : x ;
}

inline int maxabs(int x, int y)

/* returneaza maximul valorilor absolute ale lui x si y */
{
 int z= abs(x);
 int u= abs(y);

 return z < u ? u : z;
}

inline double maxabs(double x, double y)

/* returneaza maximul valorilor absolute ale lui x si y */
{
```

```
 double z= abs(x);
 double u = abs(y);

 return z < u ? u : z ;
}

main()
```
/* - citeste un sir de perechi de intregi si afiseaza, pentru fiecare pereche, maximul dintre valorile lor absolute;
- citeste un sir de perechi de numere si afiseaza, pentru fiecare pereche, maximul dintre valorile lor absolute. */

```
{
 int a,b;

 printf("sirul de perechi de numere intregi\n");

 while (scanf("%d %d",&a,&b)== 2 )
    printf("a=%d\tb=%d\tmaxabs(a,b)=%d\n",a,b,
           maxabs(a,b));
 double c,d;

 fflush(stdin); // videaza zona de intrare

 printf("sirul de perechi de numere\n");

 while(scanf("%lf %lf", &c,&d) == 2 )
    printf("c=%f\td=%f\tmaxabs(c,d)=%f\n",c,d,
           maxabs(c,d));

}
```

21. PROGRAMAREA PRIN ABSTRACTIZAREA DATELOR

Am văzut că limbajul C, şi deci şi limbajul C++, suportă stilurile de programare *procedurală* şi programare *modulară*.

Programarea procedurală (vezi paragraful 4.14.) are la bază utilizarea procedurilor a căror echivalent în limbajele C şi C++ sînt funcţiile.

Programarea procedurală este cel mai vechi stil de programare.

Mai tîrziu, pe măsură ce complexitatea programelor a crescut, a apărut ideea de a descompune problemele în subprobleme mai simple care la rîndul lor pot fi descompuse în altele mai simple şi aşa mai departe. În felul acesta se ajunge la o descompunere arborescentă a problemei date în subprobleme mai simple. Programarea subproblemelor devine o problemă mai simplă şi fiecare subproblemă are o anumită independenţă faţă de celelalte subprobleme. De asemenea, interfaţa ei cu celelalte subprobleme este limitată şi bine precizată prin procesul de descompunere a problemei iniţiale. De obicei, programarea unei subprobleme, componentă a descompunerii arborescente a problemei iniţiale, conduce la realizarea unui număr relativ mic de funcţii.

Aceste funcţii pot prelucra în comun anumite date. Unele dintre ele sînt independente de funcţiile realizate pentru alte subprobleme componente ale descompunerii arborescente. Altele realizează chiar interfaţa cu subproblemele învecinate.

Despre funcţiile obţinute în urma programării unei subprobleme se obişnuieşte să se spună că sînt *înrudite*. De obicei, aceste funcţii, împreună cu datele pe care le prelucrează, se păstrează într-un fişier şi se compilează *independent*.

O colecţie de funcţii înrudite, împreună cu datele pe care le prelucrează în comun formează un *modul*. În felul acesta, problema iniţială se realizează printr-un program alcătuit din module.

Programarea modulară are la bază elaborarea programelor pe module.

O parte din datele utilizate în comun de funcţiile modulului, sau chiar toate datele modulului, nu sînt necesare şi în alte module. Aceste date pot fi *protejate* sau cum se mai spune, *ascunse în modul*.

Limbajul C şi deci şi C++, permite ascunderea datelor în modul folosind date care au clasa de memorie *static* (vezi paragraful 5.2.).

Mai mult decît atît, pot fi declarate şi funcţiile ca statice şi atunci ele vor fi ascunse în modul (nu pot fi apelate din afara modulului). Ascunderea funcţiilor în modul se face pentru acele funcţii care nu se utilizează la realizarea interfeţei modulului cu celelalte module.

Ascunderea datelor şi funcţiilor în module permite protejarea datelor şi preîntîmpină utilizarea eronată a funcţiilor.

În capitolul 7 se dă un exemplu de modul prin care se implementează o stivă pentru numere întregi.

În capitolul respectiv, stiva este implementată printr-un *tablou* cu ajutorul declaraţiei:

```
static int stack[MAX];
```

unde *MAX* este o constantă simbolică definită în prealabil şi care stabileşte dimensiunea maximă a stivei. Numerele de tip *int* puse pe stivă sînt păstrate ca elemente ale tabloului *stack* şi ele ocupă zona:

```
stack[0],stack[1], ... ,stack[next-1]
```

Elementul stack[0] se află la baza stivei, iar stack[next-1] se află în vîrful ei.

Variabila *next* defineşte atît poziţia vîrfului stivei (elementul stack[next-1]), cît şi elementul liber al tabloului de cel mai mic indice.

Iniţial *next* are valoarea zero, deoarece stiva fiind vidă, stack[0] este elementul liber de cel mai mic indice. Variabila *next* se declară astfel:

```
static int next = 0;
```

În felul acesta, cele două date (*stack* şi *next*) utilizate pentru a implementa o stivă pentru gestionarea numerelor de tip *int* sînt ascunse în modul, avînd clasa de memorie *static*.

Gestiunea lor se face prin funcţii globale accesibile din orice modul al programului. Astfel, în exerciţiul 7.1. s-au definit funcţiile:

push - Pune în vîrful stivei un număr de tip *int*.

pop - Scoate numărul de tip *int* din vîrful stivei.

top - Permite acces la numărul de tip *int* aflat în vîrful stivei.

clear - Videază stiva.

empty - Returnează valoarea 1 dacă stiva este vidă şi zero în caz contrar.

full - Returnează valoarea 1 dacă stiva este plină şi zero în caz contrar.

Datele *stack* şi *next* fiind ascunse în modul, utilizatorul nu are acces direct la ele din alte module. De aceea, ele nu pot fi deteriorate accidental. Utilizatorul are acces numai la elementul din vîrful stivei. Acesta poate fi scos din stivă (funcţia *pop*) sau poate fi folosit lăsîndu-l pe stivă (funcţia *top*). De asemenea, utilizatorul poate pune un număr pe stivă numai în poziţia următoare celei corespunzătoare elementului din vîrful ei (funcţia *push*).

Menţionăm că stiva implementată în acest fel nu este chiar o stivă *veritabilă*, deoarece stiva, prin natura ei, este o structură dinamică. În cazul de faţă, este rezervată o zonă fixă pentru stivă deoarece tablourile din limbajul C nu sînt dinamice.

În capitolul 11 se dă o implementare pentru stive cu ajutorul listelor simplu înlănţuite. Acestea sînt structuri de date dinamice ceea ce permite ca şi stivele implementate cu ajutorul lor să fie de natură dinamică.

În acest caz, un element al stivei este o dată de un tip definit de utilizator şi anume:

typedef struct *tnod* {
  declaratii
  struct tnod **urm*;
}*TNOD*;

În acelaşi capitol, pentru gestiunea listelor simplu înlănţuite s-au folosit variabilele globale *prim* şi *ultim* care se definesc ca fiind pointeri spre tipul *TNOD*.

La implementarea stivelor printr-o listă simplu înlănţuită, operaţiile *push* şi *pop* se definesc cu ajutorul funcţiilor *iniprim* şi *spn* (vezi paragraful 11.1.3.1. şi respectiv 11.1.4.1.).

Vidarea stivei (operaţia *clear*) se realizează folosind funcţia *sterglist* definită în 11.1.5.

În mod analog, se pot defini simplu şi alte funcţii pentru a realiza accesul la elementul din vîrful stivei (corespondentul pentru *top*) sau a stabili starea stivei (vidă sau nu- corespondentul pentru *empty*) etc.

Funcţiile de gestiune a stivei pot fi rescrise în aşa fel încît să nu fie nevoie decît de variabila *prim*.

Se poate realiza şi în acest caz un modul cu ajutorul căruia să se ascundă datele de implementare ale stivei. În acest scop nu avem decît să declarăm variabila *prim* ca statică:

static *TNOD* **prim*;

Iniţial, variabila *prim* are valoarea zero (pointerul nul). Aceasta deoarece iniţial stiva este vidă.

Stivele implementate ca mai sus nu sînt convenabile dacă într-un program este nevoie să se folosească, în acelaşi timp, mai multe stive pentru date de acelaşi tip.

Într-un program se pot defini şi utiliza oricîte date pentru tipuri predefinite. De exemplu, putem defini date de tip *int* prin declaraţii de forma:

int *nume_1,nume_2, ... ,nume_n*;

Cu astfel de date se pot realiza diferite operaţii predefinite, cum sînt cele patru operaţii, operaţii de comparaţie, operaţii logice etc.

În cazul tipurilor definite de utilizator se defineşte modul de reprezentare al datelor, dar nu şi operaţiile care se realizează asupra datelor respective. De aici rezultă simplitatea în prelucrarea datelor de tipuri predefinite în comparaţie cu tratarea datelor ale căror tipuri sînt definite de utilizator.

Pentru a putea utiliza simplu, într-un program, mai multe stive pentru date de un acelaşi tip, ar fi necesar să putem defini tipul *stivă*, înţelegînd prin aceasta că se definesc atît reprezentarea datelor de tip stivă cît şi operaţiile admise asupra datelor de tip stivă.

Tipul utilizator folosit în limbajul C nu stabileşte nici o legătură între reprezentarea datelor de un anumit tip şi operaţiile executabile asupra datelor de tipul respectiv.

De exemplu, în paragraful 10.3., se defineşte tipul utilizator *COMPLEX* ca mai jos:

```
typedef struct {
    double real;
    double imag;
}COMPLEX;
```

Prin această declaraţie se defineşte reprezentarea datelor de tip *COMPLEX* şi anume, o astfel de dată este o pereche ordonată de numere, fiecare de tip *double*.

Se pot declara date de tip COMPLEX ca şi date de tip *int*, *long* sau orice alt tip predefinit. De exemplu, folosind declaraţia:

```
COMPLEX z;
```

înseamnă că *z* este o dată care se compune din două numere de tip *double*. Această declaraţie este analogă cu declaraţia:

```
int i;
```
prin care se stabileşte că *i* este o dată de tip *int*. Cum tipul *int* are predefinite şi operaţiile asupra datelor de acest tip, rezultă că declaraţia de mai sus, precizează atît reprezentarea lui *i* (reprezentarea în binar prin complement faţă de 2 pe 16 biţi) cît şi operaţiile posibile asupra lui *i*. De exemplu, expresiile de mai jos sînt corecte:

```
i+10
i/123
i++
etc.
```

În cazul datei *z*, este precizată numai reprezentarea lui *z* şi accesul la componentele sale:

z.real - Acces la prima componentă a lui z.

z.imag - Acces la a doua componentă a lui z.

La definirea tipului *COMPLEX* nu se precizează nimic asupra operaţiilor admisibile pentru datele de acest tip.

Utilizatorul poate defini astfel de operaţii prin intermediul funcţiilor. De exemplu, pentru adunarea numerelor complexe se poate defini funcţia:

```
void adcomplex(COMPLEX *a,COMPLEX *b,COMPLEX *c)
{
 c -> real = a -> real + b -> real;
 c -> imag = a -> imag + b -> imag;
}
```

În felul acesta, dacă u,v,w sînt declarate ca mai jos:

```
COMPLEX u,v,w;
...
```

atunci apelul:

```
adcomplex(&u,&v,&w);
```

realizează suma dintre numerele complexe *u* şi *v*, iar *w* devine egal cu suma respectivă, ceea ce în notaţie matematică se scrie astfel:

```
w = u+v
```

În mod analog, prin intermediul funcţiilor, se pot realiza şi alte operaţii cu numere complexe (vezi exerciţiile 10.1,10.2, 10.12, 10.13, 10.14, 10.15, 10.16, 10.17).

Prin realizarea acestor funcţii nu se precizează că ele definesc operaţiile asupra datelor de tip *COMPLEX* şi că alte operaţii sînt interzise.

Un prim pas, în încercarea de a apropia modul de tratare al tipurilor utilizator de cel al tipurilor predefinite, a fost acela de a preciza, la definirea tipului utilizator, nu numai reprezentarea datelor tipului respectiv, ci şi funcţiile care definesc operaţii cu datele de tipul respectiv.

Acest lucru s-a obţinut simplu prin enumerarea prototipurilor funcţiilor respective împreună cu definirea reprezentării datelor.

De exemplu, pentru tipul *complex* putem enumera funcţiile care realizează operaţiile cu datele de tip *complex*, ca mai jos:

```
struct complex {   // reprezentarea tipului
  double real;
  double imag;

 //functiile prin care se realizeaza operatiile asupra numerelor
 // complexe

  double modul();   // modulul numarului complex
  double arg();     // argumentul numarului complex
  void neg();       // real = -real si imag = -imag

  // real = a.real+b.real; imag = a.imag+b.imag

  void adcomplex(complex a,complex b);

  // real = a.real-b.real; imag = a.imag-b.imag

  void sccomplex(complex a,complex b);
}; // sfirsit struct complex
```

Funcţiile enumerate ca mai sus, în declaraţia *complex*, se numesc funcţii *membru*.

Prin aceasta nu se realizează facilităţi deosebite deoarece enumerarea funcţiilor membru indică, pentru programator, doar faptul că funcţiile respective sînt strîns legate de datele componente ale structurii respective.

Programatorul poate defini funcţii suplimentare care să realizeze şi alte operaţii asupra datelor structurii.

De exemplu, funcţia de prototip:

int pcitcomplex(complex ****a*);**

care citeşte partea reală şi cea imaginară a numărului *complex*, spre care pointează *a*, poate fi definită şi utilizată pentru numere de tip complex,deşi

ea nu a fost enumerată în declaraţia *struct* de mai sus.

Datele de tip complex se definesc în mod obişnuit:

```
complex z;
```

Aceasta înseamnă că *z* este o dată care are două componente:

real

şi

imag

de tip *double*.

Funcţiile membru care sînt foarte simple pot fi definite în cadrul declaraţiei *struct* înlocuind prototipul prin *antetul* şi *corpul* funcţiei respective.

De exemplu, prototipul:

```
double modul();
```

din definiţia tipului complex de mai sus, poate fi înlocuit cu definiţia funcţiei *modul*:

```
double modul()
{
 return sqrt(real*real+imag*imag);
}
```

Funcţiile membru, *definite* în interiorul declaraţiilor *struct*, se realizează în mod implicit prin expandare *inline* (vezi 20.14). De aceea, ele nu vor fi definite în interiorul declaraţiei *struct* decît dacă sînt foarte simple.

Funcţiile membru, definite în afara declaraţiei *struct*, se pot expanda *inline* dacă se declară explicit acest lucru. Deci, funcţia *modul* de mai sus, poate fi definită *inline* în afara declaraţiei *struct* dacă se declară explicit ca fiind o funcţie *inline*.

Funcţia membru *arg* este destul de complexă (are peste 3 instrucţiuni) şi de aceea ea nu va fi expandată *inline*. Deci, pentru ea se indică prototipul în declaraţia tipului *complex* şi se defineşte în afara declaraţiei respective ca o funcţie obişnuită.

O funcţie membru care se defineşte în afara declaraţiei tipului a cărui membru este, se califică cu numele tipului prin intermediul operatorului de rezoluţie(::). Astfel, numele funcţiei din antet se înlocuieşte cu:

nume_tip::nume_funcţie_membru

De exemplu, funcţia membru *arg* a tipului *complex*, definit mai sus, are antetul:

```
double complex::arg()
{
 ...
}
```

Această convenţie se aplică şi pentru funcţiile *inline* care sînt definite în afara definiţiei tipului pentru care ele sînt funcţii membru. De exemplu, funcţia *modul* se poate defini ca funcţie *inline* în afara definiţiei tipului *complex* astfel:

```
inline complex::modul()
{
 return sqrt(real*real+imag*imag);
}
```

Accesul la componentele date ale unui tip, numite şi *date membru* ale tipului respectiv, se realizează în mod obişnuit folosind operatorii punct (.) şi săgeată (->).

Fie, de exemplu, declaraţiile:

```
complex z;
complex *p;
```

La datele membru ale lui *z* ne referim prin nume calificate:

```
z.real
```

şi

```
z.imag.
```

În mod analog, la datele membru ale numărului complex spre care pointează *p* ne referim prin:

```
p -> real
```

şi

```
p -> imag.
```

În continuare, ne interesează apelul funcţiilor membru. O funcţie membru se poate apela numai pentru o dată de tipul pentru care funcţia respectivă este membru.

La apelul unei astfel de funcţii se folosesc aceeaşi operatori ca în cazul datelor membru, adică punctul şi săgeata.

Astfel, pentru calculul modulului numărului *z*, declarat ca mai sus, se va

utiliza apelul:

```
z.modul()
```

În mod analog, pentru a calcula modulul numărului complex spre care pointează *p* se va utiliza apelul:

```
p -> modul()
```

Din aceste apeluri se vede că în cazul funcţiilor membru nu este nevoie de un parametru pentru a avea acces la data curentă prelucrată de funcţia apelată.

În cazul funcţiilor obişnuite, funcţia are acces la data respectivă dacă aceasta se transferă printr-un parametru sau este o dată globală.

De exemplu, funcţia *dmodul* pentru calculul modulului unui număr de tip *COMPLEX* a fost definită în exerciţiul 10.1, ca mai jos:

```
double dmodul(ÇOMPLEX *z)
{
 return sqrt(z -> x*z -> x+z -> y*z ->y);
}
```

unde tipul *COMPLEX* se defineşte astfel:

```
typedef struct {
    double x;
    double y;
} COMPLEX;
```

Tipul *COMPLEX* definit în acest fel, nu are funcţii membru.

Dacă se foloseşte declaraţia:

```
COMPLEX a;
```

atunci pentru calculul modulului numărului complex *a* se va utiliza apelul:

```
dmodul(&a)
```

Parametrul formal *z* al funcţiei *dmodul* este un pointer spre data de tip *COMPLEX* pentru care se calculează modulul. El se utilizează în corpul funcţiei pentru a avea acces la componentele parametrului de la apel.

În cazul apelului unei funcţii membru există în mod implicit un astfel de parametru. Numele lui este *this*.

El are declaraţia implicită:

tip ***const** *this*;

unde:

tip - Este tipul pentru care funcţia apelată este o funcţie membru.

Valoarea lui *this* se defineşte la fiecare apel al funcţiei membru astfel:

- dacă se foloseşte apelul:

  nume_data.nume_functie(...)

  atunci *this* are ca valoare adresa datei *nume_data*;

- dacă se foloseşte apelul:

  pointer -> nume_functie(...)

  atunci *this* are aceeaşi valoare ca şi *pointer*.

Pointerul *this* se aplică în mod implicit la componentele datei.

Astfel, dacă considerăm apelul:

```
z.modul()
```

atunci *this* are valoarea implicită adresa lui z , adică *&z*.

În cazul funcţiei modul, *this* se utilizează în mod implicit. De aceea expresia:

```
real*real+imag*imag
```

în mod implicit se aplică la componentele datei spre care pointează *this*, adică expresia respectivă se interpretează astfel:

```
this -> real*this -> real+this -> imag*this -> imag
```

Se observă că utilizarea implicită a pointerului *this* simplifică mult definirea funcţiilor membru.

Pointerul *this* poate fi utilizat în mod explicit, de către programator, în funcţiile membru, dacă este nevoie să se facă referire la data spre care pointează *this*.

Funcţiile membru pot fi enumerate şi pentru tipurile utilizator introduse prin *union*. Acestea au aceleaşi facilităţi ca şi în cazul tipurilor definite cu *struct*.

Din cele de mai sus, rezultă că tipurile de date prezintă două aspecte: unul legat de *reprezentarea datelor*, ceea ce înseamnă că tipul este o mulţime de date care au aceeaşi reprezentare, iar celălalt aspect este legat de *operaţiile* care se definesc asupra mulţimii respective.

Tipurile de date definite în acest fel se numesc *tipuri abstracte de date*.

Tipurile abstracte de date au ca şi componente atît date cît şi funcţii. De

aceea, tipurile abstracte pot fi considerate ca o generalizare a tipului utilizator la definirea căruia se au în vedere numai *componentele dată*.

Un tip este considerat de B. Stroustrup ca o reprezentare concretă a unui concept. De exemplu, tipul *float* din limbajul C sau C++ defineşte o submulţime a mulţimii numerelor *raţionale* asupra cărora sînt definite diferite operaţii ca: adunare, scădere, înmulţire etc.

De obicei, orice problemă utilizează atît concepte cărora le corespund tipuri predefinite în limbajele de programare cît şi concepte cărora nu le corespund astfel de tipuri. Un exemplu simplu este *conceptul de stivă* pentru care, de obicei, nu există un tip predefinit în limbajele de programare.

Pentru astfel de concepte, utilizatorul urmează să definească tipuri noi. Acest lucru se poate realiza numai în parte în limbajul C cu ajutorul tipurilor definite de utilizator, deoarece nu se precizează operaţiile cu astfel de tipuri, ci numai *reprezentarea* lor.

Aşa cum s-a arătat mai sus, tipul definit de utilizator poate fi extins în limbajul C++ aşa încît odată cu definirea reprezentării datelor unui tip să se enumere, sau chiar definească, funcţiile cu ajutorul cărora se realizează operaţiile cu datele respective. Cu toate acestea, tipurile definite în acest fel nu reprezintă suficient de bine conceptul pentru care s-au definit.

Principalul neajuns rezultă din faptul că tipurile introduse în acest fel nu definesc o legătură strînsă între datele membru şi funcţiile membru. Astfel, datele unui astfel de tip pot fi accesate şi prelucrate şi de alte funcţii decît cele precizate la definirea tipului prin *struct* sau *union*. Aceasta înseamnă că la compilare nu se pot stabili controale în legătură cu utilizarea datelor de tipul respectiv în diferite funcţii. De aceea, utilizarea eronată a datelor, de astfel de tipuri, de obicei nu poate fi pusă în evidenţă prin compilare. Se obişnuieşte să se spună că datele de acest fel *nu sînt protejate*.

Amintim că programarea modulară permite o anumită protecţie a datelor prin ascunderea lor în modul.

Această idee a condus la noţiunea de *clasă* în limbajul C++. De data aceasta se încearcă ascunderea datelor la definirea tipului abstract. Astfel, datele membru (o parte sau în totalitate) pot fi protejate interzicînd accesul la ele prin funcţii care nu sînt funcţii membru ale tipului respectiv.

De asemenea, o parte din funcţiile membru pot fi şi ele protejate, neputînd fi apelate decît numai de către funcţii membru ale tipului respectiv.

Protecţiile de acest fel se realizează cu ajutorul claselor şi ele se definesc folosind *modificatori de protecţie*. Aceştia sînt trei: *private*, *protected* şi *public*.

Modificatorul *public* aplicat la o dată sau funcţie membru face ca, componenta respectivă să nu fie protejată şi ea poate fi utilizată fără restricţii în program. Primii doi modificatori (*private* şi *protected*) asigură protecţia datei sau funcţiei membru. Efectul fiecăruia, din aceşti modificatori, va fi explicat mai tîrziu. În momentul de faţă amintim că modificatorii se scriu urmaţi de două puncte (

- private:
- public:
- protected:).

O *clasă* se defineşte printr-o declaraţie de clasă. Aceasta are acelaşi format ca şi declaraţia *struct*, schimbînd cuvîntul *struct* cu *class*. Într-o astfel de definiţie, modificatorul *private* este implicit. Tipul *complex*, definit mai sus prin *struct*, îl definim printr-o clasă astfel:

```
class complex {
  double real;
  double imag;
  public:
   double modul()
   {
       return sqrt(real*real+imag*imag);
   }
   double arg();
   void adcomplex(complex a,complex b);
   void sccomplex(complex a,complex b);
};
```

Datele membru *real* şi *imag* sînt protejate prin modificatorul implicit *private*. Aceasta înseamnă că la ele nu au acces decît funcţiile membru ale clasei, adică funcţiile *modul*, *arg*, *adcomplex* şi *sccomplex*.

De exemplu, nu este posibil să atribuim valori componentelor *real* şi *imag*, să afişăm valorile lor etc.

În acest caz se realizează protecţia datelor respective. Deci, dacă vom defini o dată de tipul complex:

```
complex z;
```

atribuirile:

```
z.real = 1;
z.imag = -1;
```

sînt eronate şi vor fi semnalate la compilare.

Funcţiile membru ale clasei *complex* sînt toate publice (un modificator de protecţie este valabil din punctul apariţiei lui pînă la sfîrşitul definiţiei clasei sau pînă în punctul apariţiei unui alt modificator de protecţie).

Aceasta înseamnă că ele se pot folosi în tot programul pentru date de tip complex.

Tipurile definite prin *struct* şi *union* au toate componentele *publice*. De aceea, despre *struct*, se obişnuieşte să se spună că este o *clasă* cu *toate* componentele *publice*.

Clasele asigură protecţia datelor şi a funcţiilor membru la un nivel superior faţă de module. Ele permit definirea simplă de tipuri noi, care se numesc *tipuri abstracte de date*. Tipurile introduse cu ajutorul claselor se apropie de cele predefinite şi de aceea se pot defini şi utiliza simplu date de tipuri abstracte în comparaţie cu facilităţile oferite prin module.

Aşa cum s-a arătat mai sus, modulele nu oferă o soluţie simplă cînd utilizăm în program, în acelaşi timp, mai multe date de tipul celor ascunse într-un modul.

Clasa oferă o soluţie simplă pentru astfel de situaţii aşa cum se va vedea în continuare şi aceasta se realizează pe baza apropierii modului de utilizare al tipurilor abstracte de a celor predefinite.

B. Stroustrup afirmă că, în principiu, clasele permit să se introducă tipuri noi de date pentru care accesul este limitat printr-un set de funcţii bine precizat.

Programarea bazată pe utilizarea tipurilor abstracte de date se numeşte *programare prin abstractizarea datelor*. Ea este superioară programării modulare asigurînd protecţia datelor la un nivel calitativ superior care rezultă din facilităţile existente în definirea şi utilizarea tipurilor abstracte de date.

Întrucît tipurile abstracte de date se definesc în limbajul C++ prin intermediul claselor, rezultă că la baza programării prin abstractizarea datelor se află noţiunea de clasă.

În principiu, programatorul trebuie să stabilească clar conceptele de care are nevoie pentru rezolvarea problemei concrete şi apoi conceptele respective se realizează prin intermediul claselor.

La stabilirea şi implementarea conceptelor necesare rezolvării unei probleme se realizează un proces de abstractizare care priveşte reprezentarea datelor şi prelucrarea acestora. Aceasta justifică termenul de *programare prin abstractizarea datelor*.

Ca orice proces de abstractizare, şi cel legat de abstractizarea datelor

presupune un proces de generalizare care se realizează prin neglijarea aspectelor particulare.

Conceptele realizate prin clase au o parte *protejată* şi una *neprotejată*. Partea protejată exprimă partea specifică, proprie conceptului exprimat prin clasa respectivă şi ea are rolul de a asigura *implementarea* clasei (conceptului). Partea neprotejată, numită şi *publică*, reprezintă *interfaţa* clasei (conceptului) cu celelalte clase (concepte) ale programului. Implementarea clasei este neglijată în relaţia (utilizarea) clasei cu celelalte clase din program. În felul acesta, implementarea unei clase poate fi schimbată fără a schimba utilizarea ei în restul programului.

Exerciţii:

21.1 Se consideră tipul *complex*, definit ca mai jos, cu ajutorul lui *struct*. Se cere să se definească funcţiile membru ale tipului respectiv. Funcţia *arg* se defineşte la fel ca în exerciţiul 10.2.

FIŞIERUL BXXI1

```
#ifndef __MATH_H
#include <math.h>
#define __MATH_H
#endif

struct complex {
 double real;
 double imag;

 double modul()  //returneaza modulul numarului complex
 {
    return sqrt(real*real + imag*imag);
 }

 double arg();  //returneaza argumentul numarului complex

 void adcomplex(complex *a,complex *b) //calculeaza a+b
 {
    real = a -> real + b -> real;
    imag = a -> imag + b -> imag;
 }

 void sccomplex(complex *a,complex *b) //calculeaza a-b
 {
    real = a -> real - b -> real;
```

```
    imag = a -> imag - b -> imag;
  }

  void mulcomplex(complex *a,complex *b)
  // calculeaza a*b
  {
    real = a -> real * b -> real - a ->
           imag * b -> imag;
    imag = a -> real * b -> imag + a ->
           imag * b -> real;
  }

  int divcomplex(complex *a,complex *b);
  // calculeaza a/b
};

#ifndef __PI
#define PI 3.14159265358979
#define __PI
#endif

double complex :: arg()
// returneaza argumentul numarului complex
{
 double a;

 if(real == 0 && imag == 0)
    return 0.0;
 if(imag == 0)
    if(real > 0)
       return 0.0;
    else        // imag = 0 si real < 0
       return PI;
 if(real == 0)
    if(imag > 0)
       return PI/2;
    else       // real = 0 si imag < 0
       return (3*PI)/2;

 // real != 0 si imag != 0
 a = atan(imag/real);
 if(real < 0)   // real < 0 si y != 0
    return a+PI;
```

```
else        //real > 0
    if(imag < 0 )    //real > 0 si imag < 0
        return 2*PI + a;

 //real > 0 si imag > 0
 return a;

} //sfirsit arg

int complex :: divcomplex (complex *a,complex *b)
//calculeaza a/b; returneaza 0 la impartirea cu zero
{
 double d;
 d = b -> real * b -> real + b -> imag * b -> imag;

 if(d == 0.0)
    return 0;
 real = ( a -> real * b -> real + a -> imag * b
          -> imag)/d;
 imag = ( a -> imag * b -> real - a -> real * b
          -> imag)/d;
 return 1;
}
```

21.2 Să se scrie un program care citeşte trei numere complexe *a*, *b*, *c* care sînt coeficienţii ecuaţiei:

a*x*x+b*x+c = 0

rezolvă şi afişează rădăcinile ecuaţiei respective.

PROGRAMUL BXXI2

```
#include <stdio.h>
#include <stdlib.h>
#include "BXXI1.CPP"
#include "BX18.CPP"   //citeste un numar de tip double

int citcomplex(complex *a)
/* - citeste partea reala si cea imaginara a unui numar complex;
   - returneaza:
        0 - la sfirsit de fisier;
        1 - altfel.
*/
```

```
{
 if(pcit_double("partea reala: ", &a -> real) == 0)
    return 0;
 if(pcit_double("partea imaginara: ",
        &a -> imag) == 0)
    return 0;
 return 1;
}

main()
/* - citeste pe a,b,c - numere complexe; rezolva si
   - afiseaza radacinile ecuatiei:
             a*x*x+b*x+c = 0*/
{
 complex a,b,c,x1,x2,bp,temp,aa,temp1;
 char er[] = "s-a tastat EOF\n";
 double r,argument;
 complex patru = {4.0,0.0};

 // se citesc coeficientii

 if(citcomplex(&a) == 0){
    printf(er);
    exit(1);
 }
 if(citcomplex(&b) == 0){
    printf(er);
    exit(1);
 }
 if(citcomplex(&c) == 0){
    printf(er);
    exit(1);
 }
 if(a.real == 0 && a.imag == 0 && b.real == 0 &&
    b.imag == 0 && c.real == 0 && c.imag == 0) {

    // a = b = c = 0
    printf("ecuatie nedeterminata\n");
    exit(0);
 }
 if(a.real == 0 && a.imag == 0 && b.real == 0 &&
    b.imag == 0 ) {
    printf("ecuatia nu are solutie\n");
```

```
        exit(1);
    }
    if(a.real == 0 && a.imag == 0){
        printf("ecuatie de gradul intii\n");
        x1.real = -c.real;
        x1.imag = -c.imag;
        x2.divcomplex(&x1,&b);
        printf("x=%g+i*(%g)\n",x2.real,x2.imag);
        exit(0);
    }

    // temp = b*b - 4*a*c
    bp.mulcomplex(&b,&b);
    temp.mulcomplex(&patru,&a);
    temp1.mulcomplex(&temp,&c);
    temp.sccomplex(&bp,&temp1);

    /* temp = real + i * imag
       r = modul(temp);
       argument = arg(temp)
    */
    r = temp.modul();
    argument = temp.arg();

    /* temp = r*(cos(argument)+i*sin(argument))
       sqrt(temp) = sqrt(r)*(cos(argument/2)+i*sin(argument/2))
    */
    r = sqrt(r);
    argument = argument/2;

    // temp = sqrt(b*b-4*a*c)
    temp.real = r*cos(argument);
    temp.imag = r*sin(argument);

    // aa = a + a
    aa.adcomplex(&a,&a);

    // x1 = (-b+temp)/aa
    bp.real = -b.real;
    bp.imag = -b.imag;  // bp = -b

    // temp1 = -b+temp
    temp1.adcomplex(&bp,&temp);
```

```
    x1.divcomplex(&templ,&aa);

    // x2 = (-b-temp)/aa
    templ.sccomplex(&bp,&temp);
    x2.divcomplex(&templ,&aa);
    printf("x1 = %g + i(%g)\n",x1.real,x1.imag);
    printf("x2 = %g+i(%g)\n",x2.real,x2.imag);
}
```

22. CLASE

La programarea problemelor complexe intervin concepte noi care nu pot fi exprimate simplu prin tipuri predefinite de date.

De obicei, orice limbaj de programare pune la dispoziţia programatorului un număr de tipuri predefinite, care însă, în mod frecvent, nu corespund tuturor conceptelor necesare programului. Astfel de concepte se implementează în limbajul C++ prin intermediul claselor.

O clasă defineşte un tip abstract de date.

Prin *tip abstract de date* înţelegem o mulţime de date care au o aceeaşi reprezentare şi pentru care este definit setul de operaţii care se pot executa asupra elementelor mulţimii respective. Ca exemple de concepte care nu corespund unor tipuri predefinite din limbajul C++ amintim:

- şir de caráctere;
- număr complex;
- listă;
- arbore;
- etc.

Pentru fiecare din aceste concepte se poate defini un tip abstract de date prin intermediul claselor.

Din definiţia tipului abstract de date rezultă că acesta are 2 părţi, o parte care defineşte reprezentarea datelor tipului respectiv şi o parte care defineşte operaţiile asupra datelor respective.

Partea care defineşte reprezentarea datelor este formată din componente care sînt date de diferite tipuri. Aceste componente se numesc *date membru*.

Partea care defineşte operaţiile asupra datelor tipului respectiv conţine funcţii numite *funcţii membru*.

Prima încercare de a face legătura dintre reprezentarea datelor unui tip şi operaţiile asupra datelor respective a condus la extinderea construcţiei *struct*, aşa cum s-a văzut în capitolul precedent. Astfel, construcţia *struct* permite definirea reprezentării datelor tipului care se introduce de utilizator (*date membru*), precum şi enumerarea funcţiilor care definesc operaţii cu datele respective (*funcţii membru*).

Acest fapt nu este încă suficient pentru implementarea tipurilor

abstracte de date deoarece nu se asigură protejarea datelor membru. Ele pot fi accesate direct şi de către alte funcţii decît funcţiile membru.

Lipsa unei protecţii a datelor, face ca tipurile de date introduse prin construcţia *struct* să nu poată fi supuse unor controale cu privire la operaţiile care se execută asupra lor.

De aceea, pasul următor în implementarea tipurilor abstracte de date este acela de a introduce protejarea datelor şi funcţiilor membru. Acest lucru a condus la noţiunea de *clasă*.

Protejarea datelor şi funcţiilor membru se realizează utilizînd *modificatorii de protecţie*:

- **private;**
- **protected;**
- **public.**

Aceşti modificatori sînt urmaţi de două puncte. Modificatorii *private şi protected* protejează elementele (date şi funcţii membru) aflate în domeniul lor de acţiune.

Domeniul de acţiune al unui modificator de protecţie ţine din punctul în care este scris modificatorul respectiv şi pînă la sfîrşitul definiţiei care îl conţine sau pînă la un alt modificator de protecţie.

Membrii din domeniul de acţiune al modificatorului *public* nu sînt protejaţi şi ei pot fi folosiţi fără restricţii în tot programul unde ei sînt "vizibili".

În mod implicit, membrii unei clase sînt protejaţi ca şi în cazul în care s-ar afla în domeniul de acţiune a lui *private*. Despre un astfel de membru vom spune că este *privat*. El poate fi utilizat numai de către o funcţie membru. Vom spune că accesul la un astfel de membru nu se face direct, ci *indirect* prin intermediul funcţiilor membru. Nu acelaşi lucru este valabil pentru tipurile definite cu ajutorul construcţiei *struct*.

În acest caz, membrii *dată* sau *funcţie* sînt în mod implicit publici, deci la ei putem face acces direct. Aceasta explică de ce în programul din exerciţiul 21.2. se pot folosi direct (în afara funcţiilor membru) expresii de forma:

a.real

şi

a.imag

unde:

a - Este o variabilă de tip *complex*.

Într-adevăr, construcţia *struct* nu protejează datele membru *real* şi *imag*, deci ele pot fi folosite în mod obişnuit, la fel ca în limbajul C. De aceea, compilatorul nu poate realiza nici un control asupra operaţiilor realizate cu datele de tip *complex*, definite cu ajutorul construcţiei *struct*, ca în exerciţiul 21.1. Cu totul alta este situaţia dacă definim tipul *complex* cu ajutorul unei clase.

Menţionăm că pentru a defini o clasă se utilizează acelaşi format ca şi în cazul construcţiei *struct*.

La definirea tipului *complex* cu ajutorul clasei vom proteja datele membru, adică *real* şi *imag*.

Formatul clasei pentru tipul *complex* va fi:

```
class complex {
    // date membru private
    double real;
    double imag;

    // functii membru
    ...
};
```

În continuare, putem defini date de tip complex în mod obişnuit. De exemplu:

```
complex a;
```

defineşte pe *a* de tip complex. În acest caz, *a* este o dată care are două componente: *real* şi *imag* , ambele de tip *double* dar ele sînt private şi deci nu putem avea acces direct la ele. De aceea, instrucţiunile de mai jos sînt interzise în afara funcţiilor membru:

```
a.real = 1;
a.imag = 2;
printf("a=%g+i*(%g)\n",a.real,a.imag);
```

Mai mult decît atît, datele de acest tip nici nu pot fi iniţializate în mod obişnuit. Deci, o declaraţie de forma:

```
complex a = {1,-1};
```

este interzisă.

Deoarece singurul acces posibil la datele membru *real* şi *imag* este prin intermediul funcţiilor membru, rezultă că este nevoie de funcţii membru care să realizeze nu numai operaţiile obişnuite cu numere complexe (modul,

argument, adunare, scădere, înmulţire şi împărţire), ci şi alte operaţii cum sînt:

- iniţializare;
- atribuire;
- afişare.

Este necesar ca toate aceste funcţii membru să fie publice. Rezultă că o primă variantă pentru definirea tipului *complex* ar putea fi clasa de mai jos:

```
class complex {
 // date membru private

     double real;
     double imag;

 public:          // functii membru publice

     void atribuire(double x = 0,double y = 0)
     // real+i*imag = x+i*y
     {
        real = x;
        imag = y;
     }

     double retreal()
     // returneaza partea reala a numarului complex
     {
        return real;
     }

     double retimag()
     //returneaza partea imaginara a numarului complex
     {
        return imag;
     }

     void afiscomplex(char *format)
     //afiseaza numarul complex conform formatului
     // definit de pointerul format
     {
        printf(format,real,imag);
     }
```

```
void adcomplex(complex *x,complex *y)
// calculeaza x+y
{
    real = x -> real + y -> real;
    imag = x -> imag + y -> imag;
}

void negcomplex(complex *x)
// calculeaza -x
{
    real = -x -> real;
    imag = -x -> imag;
}

void sccomplex(complex *x,complex *y)
// calculeaza x-y
{
    real = x -> real-y -> real;
    imag = x -> imag - y -> imag;
}

void mulcomplex(complex *x,complex *y)
// calculeaza x*y
{
    real = x -> real*y -> real-x -> imag*y
           -> imag;
    imag = x -> real*y -> imag+x -> imag*y
           -> real;
}

int divcomplex(complex *x,complex *y);
/* - calculeaza x/y;
   - returneaza:
           0 - la impartirea cu zero;
           1 - altfel.
*/

}; // sfirsitul definitiei clasei complex

int complex :: divcomplex(complex *a,complex *b)
/* - calculeaza a/b;
   - returneaza:
       0 - la impartirea cu zero;
```

```
    1 - altfel.
*/
{
 double d;

 d = b -> real*b -> real+b -> imag*b -> imag;
 if(d == 0.0)
     return 0;
 real = (a -> real*b -> real+a -> imag*b
         -> imag)/d;
 imag = (a -> imag*b -> real-a -> real*b
         -> imag)/d;
 return 1;
}
```

Funcţiile membru sînt funcţii *inline*, exceptînd funcţia *divcomplex*, care, conţinînd mai multe instrucţiuni decît celelalte, a fost definită în afara definiţiei clasei *complex* şi deci ea nu este o funcţie *inline*.

În antetul funcţiei *divcomplex* s-a utilizat numele:

```
complex :: divcomplex
```

construit cu ajutorul operatorului de rezoluţie. Această regulă se utilizează pentru toate funcţiile membru care nu se definesc în interiorul definiţiei clasei pentru care ele sînt funcţii membru.

O clasă defineşte un tip care a fost numit *tip abstract*.

Partea privată a unei clase defineşte modul de *implementare* al datelor de tipul abstract definit de clasa respectivă. Partea publică defineşte *interfaţa* acestor date cu restul programului. Această interfaţă conţine, de obicei, funcţii care au fost numite *funcţii membru*. Funcţiile membru se mai numesc şi *metode*.

În general, partea publică poate conţine şi date membru, dar atunci datele respective nu mai sînt protejate, lucru care pe cît posibil trebuie evitat. Aşa cum s-a subliniat mai sus, lipsa unei protecţii pentru datele membru nu permite compilatorului să controleze operaţiile realizate cu datele respective.

În cazul clasei *complex* de mai sus, programatorul nu are acces *direct* la componentele *real* şi *imag* ale datelor de tip complex. Toate operaţiile care se pot realiza cu datele de acest tip se fac numai prin intermediul funcţiilor membru care au fost declarate ca publice.

Datele ale cărui tip se definesc printr-o clasă se numesc *obiecte*. Cu alte cuvinte, un obiect în C++ este o dată de un tip abstract.

Obiectele, la fel ca şi datele predefinite, se declară printr-o declaraţie.

În forma cea mai simplă, o declaraţie de obiect are formatul:

nume_clasă listă_de_nume_de_obiecte;

unde prin *listă_de_nume_de_obiecte* înţelegem un nume sau mai multe separate prin virgulă. De asemenea, *nume_clasă* este un nume.

Exemplu:

Dacă se consideră definiţia clasei *complex* de mai sus, atunci:

```
complex z,x1,x2;
```

declară pe *z, x1* şi *x2* ca obiecte, adică date de tip *complex*.

Despre obiectele unei clase se obişnuieşte să se spună că sînt *instanţieri* ale clasei respective. Deci *z, x1* şi *x2*, sînt instanţieri ale clasei complex.

Pentru a apropia tipurile abstracte de cele predefinite, se impune ca obiectele să poată fi *iniţializate*.

Iniţializarea unui obiect este o problemă mult mai complexă decît cea a datelor obişnuite (tipuri predefinite sau definite de utilizator cu ajutorul construcţiilor *struct* sau *union*). De aceea, iniţializarea unui obiect este posibilă folosind în acest scop o funcţie membru specială. Aceasta poartă denumirea de *constructor*.

Constructorul unei clase are totdeauna *acelaşi* nume cu numele clasei.

Dacă o clasă are constructor, atunci acesta se apelează automat la instanţierea unui obiect al clasei respective.

Exemplu:

Pentru clasa *complex* definită mai sus, putem defini un constructor cu ajutorul funcţiei membru:

```
complex(double a=0,double b=0)
// constructorul clasei complex
{
 real = a;
 imag = b;
}
```

Această funcţie membru se presupune că se defineşte în interiorul definiţiei clasei *complex*. Deci, este o funcţie inline. Pentru a o defini în afara definiţiei clasei *complex* se schimbă antetul funcţiei astfel:

```
inline complex :: complex(double a,double b)
```

Pentru a iniţializa un obiect, în declaraţia obiectului, după numele lui, se includ în paranteze rotunde valorile efective ale parametrilor constructorului.

Exemplu:

```
complex z(1,2);
```

Se instanţiază obiectul *z*, de tip complex, ale cărui componente au valorile iniţiale:

```
z.real = 1
```

şi

```
z.imag = 2
```

În cazul în care există o singură valoarea de iniţializare, atunci se poate utiliza semnul de atribuire (=). De exemplu, declaraţia:

```
complex z = 123;
```

este identică cu declaraţia:

```
complex z(123);
```

Ele iniţializează obiectul *z* astfel:

- z.real = 123;
- z.imag = 0 - valoarea implicită.

La o declaraţie de forma:

```
complex z;
```

adică fără iniţializare, constructorul este apelat în mod automat şi se instanţiază obiectul *z* cu ambele componente egale cu zero (valorile implicite ale parametrilor constructorului).

Obiectele sînt *distruse* de către programator sau în mod automat.

Distrugerea obiectelor se poate realiza folosind o funcţie membru specială numită *destructor*. Un destructor are un nume aparte şi anume:

~*nume*

unde:

nume - Este numele clasei.

De asemenea, destructorul nu are parametri. El se apelează de programator cînd nu mai este nevoie de un obiect. Acest lucru este important atunci cînd obiectul a fost creat în memoria *heap*.

Destructorul se apelează automat cînd obiectul își încetează existența (vezi clasele de memorie).

Funcțiile membru se apelează calificînd numele lor cu numele obiectului pentru care se realizează apelul:

nume_obiect.nume_functie_membru(lista parametrilor efectivi)

De exemplu, dacă *z* este o instanțiere a clasei *complex* (complex z;), atunci pentru a afișa partea reală și cea imaginară a obiectului *z*, se va apela funcția membru *afiscomplex* ca mai jos:

```
z.afiscomplex("Partea reala = %g\t\
              Partea imaginara = %g\n");
```

În mod analog, dacă obiectele *z*, *z1* și *z2* sînt instanțieri ale clasei *complex*, atunci pentru a realiza atribuirea:

```
z = z1+z2;
```

se va apela funcția membru *adcomplex* astfel:

```
z.adcomplex(&z1,&z2);
```

În cazul în care se utilizează un pointer spre un obiect:

*nume_clasa *po*;

funcțiile membru se apelează printr-o construcție de forma:

po -> nume_functie_membru(lista parametrilor efectivi)

În corpul unei funcții membru, ne putem referi la obiectul pentru care a fost apelată funcția membru folosind pointerul implicit *this*. Acest pointer se utilizează în corpul funcțiilor membru în mod implicit la datele membru ale obiectului pentru care este apelată funcția membru respectivă. Cu toate acestea, programatorul poate utiliza explicit pointerul *this* pentru a se referi, în corpul unei funcții membru, la obiectul pentru care s-a apelat funcția respectivă.

În continuare, prin *obiect curent* vom înțelege obiectul pentru care s-a apelat o funcție membru, adică obiectul spre care pointează pointerul *this*.

Apelul unei funcții membru referitor la un obiect al clasei pentru care funcția respectivă este funcție membru, simplifică modul de exprimare al operațiilor care se realizează asupra obiectelor. De asemenea, acest mod de apel poate fi controlat de compilator. Orice apel neautorizat va fi semnalat și eliminat de compilator. În felul acesta, lucrul cu obiectele unei clase devine similar cu cel existent pentru datele de tipuri predefinite:

- obiectele se pot inițializa;

- asupra lor se pot face numai operaţii în prealabil definite prin funcţii membru şi care sînt controlabile prin compilator.

Protecţia datelor şi funcţiilor membru ale unei clase care nu sînt *publice* constituie un mijloc de "ascundere" a elementelor membru respective. Această "ascundere", numită şi *încapsulare* se realizează la un nivel calitativ superior faţă de ascunderea realizată cu ajutorul modulelor.

De exemplu, prin intermediul clasei *complex*, definită mai sus,se pot instanţia oricîte obiecte (numere) de tip *complex* care pot fi şi iniţializate, iar operaţiile cu ele se realizează simplu cu ajutorul funcţiilor membru şi aplicarea lor este controlată de compilator.

Mai jos, introducem tipul abstract *stiva* pentru a pune mai bine în evidenţă diferenţa dintre facilităţile obţinute pe baza utilizării claselor faţă de cele oferite de programarea modulară.

Amintim că în capitolul 7, a fost definit un modul pentru implementarea unei stive cu ajutorul unui tablou de tip *int*.

Mai jos, vom implementa tipul *stiva* tot cu ajutorul unui tablou de tip *int* care are cel mult 100 de elemente.

Modulul definit în capitolul 7 implementează o *singură* stivă. În cazul de faţă se pot instanţia oricîte obiecte de tip stivă.

```
enum Boolean {false,true};
class stiva {
    int stack[100];
    int istack;

 public:
    stiva()  //constructor
    {
        istack=0;            //la inceput stiva este vida
    }

    void push(int n);//pune pe n pe stiva

    int pop();               //scoate elementul din virful stivei
                             //se returneaza elementul din virful stivei

    int top();               //returneaza elementul din virful stivei
                             //elementul ramine pe stiva
```

```
void clear()      // videaza stiva
{
    istack=0;
}

Boolean empty()  // returneaza true daca stiva este vida
                 // si false altfel
{
    return istack == 0 ? true : false;
}

Boolean full()  // returneaza true daca stiva este plina
                // false altfel
{
    return istack == 100 ? true : false;
}

};
```

În continuare, se pot instanţia obiecte de tip *stiva*:

```
stiva stiva1, stiva2;
```

La instanţierea obiectelor *stiva1* şi *stiva2* se apelează în mod automat constructorul clasei. Acesta realizează atribuirea:

```
istack = 0;
```

În felul acesta, stivele *stiva1* şi *stiva2* sînt vide la instanţiere.

Datele membru *stack* şi *istack* sînt "încapsulate" (ascunse) în clasa *stiva*, fiind private. Ele nu pot fi accesate direct ca şi în cazul modulului definit în capitolul 7.

Stivele se pot gestiona simplu cu ajutorul funcţiilor membru. Astfel, pentru a pune un element pe stiva *stiva1*, utilizăm apelul funcţiei *push*:

```
stiva1.push(expresie)
```

În mod analog, punem un element pe stiva *stiva2*, apelînd aceeaşi funcţie *push* ca mai jos:

```
stiva2.push(expresie)
```

În felul acesta, funcţiile membru gestionează simplu oricîte stive de tip *stiva*. Acest lucru nu este posibil prin intermediul modulului descris în capitolul 7. Evident, se poate construi un modul care să gestioneze mai multe stive, dar acest lucru nu se realizează atît de simplu şi elegant ca mai

sus, prin utilizarea claselor. Acest fapt explică afirmaţia lui B. Stroustrup: dacă într-un program este suficient un *singur* exemplar al unei date de un anumit tip, atunci este suficient un modul pentru lucrul cu data respectivă. Altfel, se defineşte o clasă pentru tipul datei respective şi astfel se vor putea instanţia oricîte exemplare.

În aceasta constă esenţa saltului realizat prin trecerea de la programarea modulară la stilul programării prin abstractizarea datelor.

În rezumat, programarea prin abstractizarea datelor are la bază utilizarea tipurilor abstracte de date. În acest scop, se porneşte cu stabilirea conceptelor necesare la realizarea unui program. O parte din aceste concepte se realizează cu ajutorul tipurilor predefinite, existente în limbajul de programare utilizat (în cazul de faţă limbajul C++). Celelalte concepte se realizează sub formă de tipuri abstracte de date care se definesc în limbajul C++ cu ajutorul claselor.

Un tip abstract de dată prezintă două aspecte: unul legat de implementarea tipului şi celălalt legat de utilizarea lui, care defineşte interfaţa tipului respectiv cu restul programului.

Implementarea tipului este partea lui protejată, iar interfaţa constituie partea publică a tipului, utilizabilă în tot programul.

Despre partea protejată se spune că,conţine informaţia încapsulată în clasa care defineşte tipul respectiv.

La definirea unei clase se precizează:

- reprezentarea datelor tipului abstract;
- funcţiile care descriu operaţii cu datele tipului abstract.

Reprezentarea datelor se defineşte prin componente date numite date membru.

Funcţiile care definesc operaţii cu datele tipului abstract, se numesc funcţii membru.

De obicei, partea încapsulată în clasă conţine date membru, dar ea poate conţine şi funcţii membru.

Elementele publice sînt funcţii membru. Ele pot fi şi date membru, dar atunci acestea nu mai sînt protejate, lucru care este bine să fie evitat.

Un tip abstract de date trebuie astfel definit, încît la utilizarea lui să se facă abstracţie de implementare. Mai mult decît atît, utilizarea datelor de un tip abstract nu trebuie să fie influenţată de implementare. Aceasta înseamnă că implementarea tipului abstract poate fi oricînd schimbată, fără a schimba şi utilizarea, în program, a datelor tipului respectiv.

De exemplu, la implementarea tipului *stiva* s-a folosit un tablou de tip *int* de 100 de elemente şi variabila *istack* de tip *int*.

Schimbînd implementarea prin înlocuirea tabloului cu o listă simplu înlănţuită (vezi capitolul 11), interfaţa formată din funcţiile membru:

push, pop, top, clear, empty şi *full*

se utilizează în acelaşi mod. Evident, funcţiile se modifică, dar ele sînt apelate la fel şi au acelaşi efect. Dacă acest lucru nu este realizabil, înseamnă că tipul abstract nu a fost bine definit, deoarece la utilizarea lui nu se poate face abstracţie de implementare.

Datele de un tip abstract sînt numite obiecte ale tipului respectiv.

Obiectele se declară printr-o declaraţie asemănătoare cu declaraţia datelor pentru tipuri predefinite. Ele pot fi iniţializate la declararea lor.

Obiectele se creează, de obicei, printr-un constructor, care este o funcţie membru specială, al cărui nume coincide cu numele clasei. Constructorul unei clase se apelează automat la construirea unui obiect al clasei respective. Despre un obiect al unei clase se spune că este o instanţiere a clasei respective.

Un obiect poate fi distrus printr-o funcţie specială numită *destructor*. Numele destructorului este numele clasei precedat de caracterul "~".

Funcţiile membru pot fi definite în interiorul definiţiei clasei dacă sînt foarte simple. Aceasta deoarece, în acest caz, ele se definesc în mod implicit ca funcţii inline.

O funcţie membru definită în afara definiţiei clasei are numele din antet construit cu ajutorul operatorului de rezoluţie:

nume_clasa :: nume_functie_membru

La apelul unei funcţii membru se califică numele funcţiei cu numele obiectului pentru care se apelează funcţia.

Facilităţile obţinute cu ajutorul claselor apropie modul de tratare şi utilizare al obiectelor de cel al datelor de tipuri predefinite.

O diferenţă între utilizarea obiectelor şi a datelor de tipuri predefinite apare la scrierea expresiilor. Aşa de exemplu, dacă se consideră datele declarate ca mai jos:

```
int i,j,k;
double a,b,c;
```

Atunci se pot utiliza expresii de forma:

```
k = i+j;
c = a+b;
```

care sînt sugestive față de exprimarea adunării datelor de tip *complex* prin apelul funcției membru *adcomplex*:

```
complex z,u,v;
...
v.adcomplex(&z,&u);
```

De aceea, ar fi bine ca expresii de forma:

```
v = z+u;
```

să fie acceptate de compilator şi pentru obiecte. Această idee ne conduce la noțiunea de *supraîncărcare a operatorilor*. Într-adevăr, se poate spune că operatorul + este supraîncărcat pentru tipuri predefinite.

El se poate aplica la operanzi de tipuri numerice predefinite (*int, long, float, double, unsigned* şi *long double*). De aceea, pentru a putea scrie adunarea obiectelor complexe *z* şi *u* sub forma expresiei:

```
z+u
```

ca în cazul tipurilor numerice predefinite, este suficient ca operatorul + să poată fi supraîncărcat cu operanzi de tip complex.

Supraîncărcarea operatorilor este o facilitate care simplifică mult exprimarea operațiilor asupra obiectelor. Pe baza ei se pot scrie expresii cu obiecte la fel ca şi cu date de tipuri predefinite.

Supraîncărcarea operatorilor se realizează prin construcții asemănătoare funcțiilor dar care au un antet special în care este prezent cuvîntul *operator*.

În sfîrşit amintim că, supraîncărcarea operatorilor pentru obiecte este completată şi cu diferite conversii care se aplică la operații cu obiectele respective. În felul acesta se ajunge să utilizăm obiectele la fel de simplu ca şi datele de tipuri predefinite.

În încheiere putem afirma că, clasele sînt un instrument pentru a *crea* tipuri noi care pot fi utilizate tot aşa de bine ca şi tipurile predefinite. Ideal, tipurile noi (tipurile abstracte) nu trebuie să difere, în utilizare, de tipurile predefinite, ci numai în modul în care sînt create.

22.1. Definiția claselor

O clasă defineşte un tip abstract de date. Ea are o definiție a cărei format simplificat este indicat mai jos:

class *nume* {*lista_elementelor_membru*};

Ulterior o să vedem şi un alt format mai complex, pentru clase.

În limbajul C++, formatul de mai sus poate fi folosit şi pentru a defini tipuri noi cu ajutorul cuvintelor cheie *struct* şi *union*. Deci,în formatul de mai sus, se poate înlocui cuvîntul *class* prin *struct* sau *union*. De fapt, în cele ce urmează, vom considera că şi formatele în care cuvîntul cheie *class* se înlocuieşte prin *struct* sau *union* definesc tot clase. Diferenţa constă în aceea că, în cazul utilizării cuvîntului *class*, elementele membru în mod implicit sînt protejate prin protecţia oferită de modificatorul de protecţie *private*, iar în cazul lui *struct* şi *union*, în mod implicit elementele membru sînt neprotejate (publice).

În cazul utilizării cuvîntului *struct* se poate modifica protecţia cu ajutorul modificatorilor de protecţie. În schimb, în cazul utilizării cuvîntului *union*, elementele membru pot fi numai publice. De aceea, se obişnuieşte să se spună că *struct* şi *union* definesc clase cu elemente membru publice.

În cele ce urmează vom înţelege prin clasă de tip *struct* o clasă definită cu ajutorul lui *struct*, iar prin clasă de tip *union*, o clasă definită cu ajutorul lui *union*.

Numele aflat după *class*, *struct* sau *union* este numele tipului introdus prin definiţia respectivă. El se numeşte şi *numele clasei* şi în continuare poate fi utilizat pentru a declara (instanţia) date de tipul respectiv, date numite şi obiecte de tipul respectiv.

Lista elementelor membru poate conţine:

- declaraţii de date;
- definiţii de funcţii;
- prototipuri de funcţii;
- modificatori de protecţie.

Datele declarate într-o definiţie de clasă se numesc *date membru*.

Funcţiile definite sau pentru care este prezent numai prototipul, în definiţia clasei, se numesc *funcţii membru*.

Amintim că funcţiile definite în definiţia unei clase trebuie să fie simple, deoarece ele se definesc în mod implicit ca funcţii *inline*. Pentru funcţiile mai complexe se indică în definiţia clasei numai prototipurile funcţiilor respective şi ele se definesc ulterior. În acest caz, antetul funcţiei conţine un nume format cu ajutorul operatorului de rezoluţie:

nume_clasă :: nume_funcţie_membru

De obicei, o clasă are unul sau mai mulţi constructori precum şi un destructor. Aceştia sînt funcţii membru de nume speciale, şi anume, numele constructorilor coincide cu numele clasei (dacă sînt mai mulţi constructori, atunci aceştia sînt funcţii supraîncărcate), iar numele destructorului este numele clasei precedat de caracterul ~ (totdeauna există cel mult un destructor).

Un alt caz particular de funcţii membru sînt cele care definesc operatori şi conversii pentru obiectele clasei respective.

Ca modificatori de protecţie am văzut că se pot folosi:

- **private:**
- **protected:**

şi

- **public:**

Primii doi asigură o protecţie a datelor sau funcţiilor membru din domeniul de acţiune al lor, iar ultimul se utilizează pentru elemente membru care dorim să nu fie protejate.

Domeniul unui modificator de protecţie începe din punctul în care este scris şi pînă la sfîrşitul definiţiei clasei care îl conţine sau pînă la întîlnirea unui alt modificator de protecţie.

De obicei, datele membru sînt protejate, dar aceasta nu înseamnă că ele nu pot fi publice.

De asemenea, funcţiile membru, numite şi metode, de obicei sînt publice, dar ele pot fi şi protejate.

Clasele pot fi definite incomplet în cazul în care este nevoie să ne referim la ele. O astfel de definiţie incompletă are formatul:

- **class** *nume*;
- **struct** *nume*;

sau

- **union** *nume*;

Datele membru se declară în mod obişnuit. Dacă este prezentă o clasă de memorie, atunci aceasta poate fi numai clasa de memorie *static*.

O dată membru a unei clase nu poate fi de tipul definit prin clasa respectivă. Ea poate fi numai un pointer spre tipul respectiv sau o referinţă la tipul respectiv.

Fie definiţia:

```
class nume {
  ...
  nume *ptr ;      // corect
  . . .
  nume& ref;       // corect
  . . .
  nume obiect;     // incorect
};
```

Data membru *obiect* de tip *nume* nu este admisă. În schimb, se pot declara date de tipuri introduse prin alte clase.

Domeniul de existenţă al numelui unei clase este din punctul definirii ei pînă la sfîrşitul blocului (instrucţiunii compuse care o conţine).

De obicei, definiţia unei clase se scrie la începutul fişierului sursă în care este utilizată şi în afara oricărui bloc. Se obişnuieşte adesea să se construiască un fişier de tip *h* care să conţină definiţii de clase. Fişierul respectiv se include la începutul fiecărui fişier care utilizează definiţiile claselor respective.

22.2. Obiecte

Un obiect este o dată de un tip definit printr-o clasă. Se obişnuieşte să se spună că este o instanţiere a clasei respective.

Declaraţia de obiecte este asemănătoare cu cea pentru datele de tipuri predefinite.

În cea mai simplă formă, un obiect se declară folosind formatul:

nume_clasa nume_obiect;

De obicei, clasele au constructori care se apelează automat la întîlnirea declaraţiei de instanţiere a unui obiect al clasei respective.

Datele membru se alocă distinct la fiecare instanţiere a clasei. Deci, datele membru există în atîtea exemplare cîte obiecte au fost instanţiate. O excepţie o constituie datele membru care au clasa de memorie *static*. O dată membru de clasă de memorie *static* (numită *dată membru statică*) este o parte comună pentru toate instanţierile clasei şi există într-un singur exemplar.

Funcţiile membru sînt într-un singur exemplar oricîte instanţieri ar exista. O funcţie membru se apelează totdeauna în strînsă dependenţă cu un

obiect care este o instanţiere a clasei pentru care funcţia respectivă este funcţie membru.

Legătura dintre funcţii membru şi obiectul pentru care se face apelul se realizează folosind operatorul punct sau săgeată.

Exemplu:

Considerăm tipul *complex* definit la începutul capitolului. Fie declaraţiile:

```
complex z;
complex *pz;
char *format = "%g\t%g\n";
```

Atunci:

```
z.afiscomplex(format);
```

afişează componentele numărului complex z (în cazul de faţă 0 0 deoarece constructorul clasei are parametri impliciţi zero).

Acelaşi efect se obţine folosind secvenţa:

```
pz = &z;
pz -> afiscomplex(format);
```

Programatorul poate utiliza în mod explicit pointerul *this*, în corpul unei funcţii membru. Acesta are ca valoare adresa obiectului curent, adică a obiectului pentru care s-a făcut apelul funcţiei membru.

O excepţie de la aceste reguli o reprezintă funcţiile membru care au clasa de memorie *static*. Acestea se numesc funcţii membru statice. O astfel de funcţie poate fi apelată în două moduri:

- Ca orice funcţie membru, folosind operatorul punct sau săgeată.
- Independent de un obiect al clasei pentru care este funcţie membru. În acest caz, apelul se realizează folosind operatorul de rezoluţie, adică sub forma:

  nume_clasa :: nume_funcţie_membru_statica

  În acest caz,pointerul *this* nu mai poate fi utilizat.

Exemplu:

Fie definiţia de clasă:

```
class dc {
    int zi,luna,an;
    static int zz,ll,aa;
```

```
public:
    dc(int z=1,int l=1,int a=1995)
    {
        zi = z; luna = l; an = a;
    }
    static Boolean v_calend(dc *d);
    ...
};
```

Datele membru *statice*: *zz*, *ll*, *aa* sînt comune pentru toate instanţierile, spre deosebire de celelalte date care se alocă distinct la fiecare instanţiere. Acestea din urmă, se iniţializează cu ajutorul constructorului.

Funcţia *v_calend* verifică data calendaristică definită de datele membru *zi*, *luna* şi *an*, care sînt componente ale obiectului spre care pointează *d*, precum şi cea definită de datele membru statice *zz*, *ll* şi *aa*. Ea poate fi definită ca mai jos:

```
Boolean dc :: v_calend(dc *d)
/* verifica data calendaristica
        - zi, luna, an, ale obiectului spre care pointeaza d;
        - zz, ll, aa;
        - returneaza True daca datele sint corecte şi False altfel.
*/
{
  static tnrz[] =
   {0,31,28,31,30,31,30,31,31,30,31,30,31};
  int z,l,a;

  if(d -> an < 1600 || d -> an > 4900)
      return False;
  if(d -> luna < 1 || d -> luna > 12)
      return False;
  if(d -> zi < 1 || d -> zi > tnrz[d -> luna]+
    (d -> luna==2 && (d ->an%4==0 && d -> an%100 ||
     d ->an%400 == 0)))
      return False;
  z = dc :: zz;
  l = dc :: ll;
  a = dc :: aa;
  if(a < 1600 || a > 4900)
      return False;
  if(l < 1 || l > 12)
      return False;
```

```
if(z<1 || z>tnrz[l]+(l==2 && (a%4==0 && a%100 ||
    a%400==0)))
    return False;
 return True;
}
```

Fie instanţierea:

```
dc data_calend(29,2);
```

Pentru a valida instanţierea *data_calend* se va apela funcţia *v_calend* astfel:

```
if(dc::v_calend(&data_calend) == False){
    // data calendaristica eronata
    ...
}
else{
    // data calendaristica corecta
    ...
}
```

În acest caz, funcţia *v_calend* a fost apelată folosind operatorul de rezoluţie.

Se verifică data calendaristică definită de datele membru *zi, luna* şi *an* a instanţierii *data_calend*. Constructorul acestei clase a iniţializat obiectul *data_calend* cu valorile:

zi = 29
luna = 2
an = 1995

Datele membru statice(zz,ll,aa) există în afara instanţierilor clasei *dc*. Deşi aceste date sînt private, referirea la ele se poate face în funcţia *v_calend* care este o funcţia membru a clasei *dc*. Totuşi, referirea la ele nu se poate face direct, deoarece în acest caz pointerul *this* nu este definit, funcţia *v_calend* nefiind apelată pentru un obiect. De aceea, referirea la datele statice *zz, ll,* şi *aa* s-a făcut folosind numele clasei şi operatorul de rezoluţie:

dc :: zz, dc :: ll

şi

dc :: aa

22.3. Domeniul unui nume

Unui *nume* îi corespunde un *domeniu*. Acesta se defineşte prin declaraţia lui sau este *corpul unei funcţii* dacă el este numele unei *etichete*.

Prin *bloc* înţelegem o instrucţiune compusă. În limbajul C o declaraţie poate fi în interiorul unui bloc sau în afara blocurilor. În plus, în limbajul C++ o declaraţie poate fi şi în interiorul unei clase (în interiorul declaraţiei de clasă).

O declaraţie, care este în afara blocurilor sau a claselor şi care nu este o declaraţie de date externe, se spune că este o *definiţie*.

Pentru un nume vom distinge 3 tipuri de domenii, în funcţie de poziţia declaraţiei (definiţiei) sale. Aceste tipuri sînt: *local*, *fişier* şi *clasă*.

Un nume declarat într-un bloc are un *domeniu de tip local*. Acesta începe în punctul în care este declarat şi ţine pînă la sfîrşitul blocului respectiv. Un astfel de nume poate fi utilizat în domeniul lui, inclusiv în blocurile incluse în domeniul respectiv, dacă el nu este redeclarat în aceste blocuri.

Exemplu:

Fie instrucţiunile compuse imbricate de mai jos:

```
{
  ...
  int i;    // incepe domeniul de tip local a lui i
  ...       // nu incepe alt bloc

  i=10;     // atribuire corectă
  ...

  {
     int j;
     ...    // nu exista declaratie pentru i
     j=i+2; //expresie corecta deoarece i se afla in
            // domeniul lui si nu a fost redeclarat

     ...
  }
  ...
  {
     long n;
     long i;  // redeclararea lui i
              // in acest bloc nu se poate utiliza variabila
```

```
            // i declarata la inceput de tip int
      ...
      n = i+20;       // se utilizeaza i de tip long
      ...
   }  // in acest punct se termina domeniul lui i declarat de tip long i
   ... // nu exista redeclararea lui i
       // este valabila prima declaratie a lui i
   i=100;      // se atribuie 100 variabilei i de tip int
   ...
}  // in acest punct se termina domeniul lui i declarat int i
```

Un nume declarat în afara oricărui bloc sau declaraţie (definiţie) de clasă are un *domeniu de tip fişier*. Acest domeniu începe în punctul în care numele este definit şi ţine pînă la sfîrşitul fişierului care conţine definiţia respectivă. El poate fi utilizat în domeniul respectiv fără nici o restricţie dacă nu este redefinit în blocurile incluse în domeniul său.

Dacă un nume care are un domeniu de tip fişier este redefinit într-un bloc inclus în domeniul său, atunci el poate fi folosit,în acel bloc, dacă este precedat de operatorul de rezoluţie.

Exemplu:

```
...
int i = 100; // definitia lui i nu este inclusa in nici un bloc sau clasa;
             // are domeniu de tip fisier care incepe in acest punct
...
{
  int j;
  ...     // i nu este redeclarat pina in acest punct
  j=i+123;    // se utilizeaza i definit mai sus
  ...
  long i; // redeclarare a lui i
          // incepe un domeniu de tip local pentru i
  j=i-123; // se utilizeaza i declarat prin long i
  ...
  j=j+ :: i // se utilizeaza i declarat prin int
  ...
}
...
// sfirsit fisier: se termina domeniul lui i declarat prin int i
```

O clasă are o declaraţie (definiţie) care defineşte un *tip abstract de date*. *Numele acestui tip* este considerat totodată ca fiind *numele unei clase*.

Numele unei clase are un domeniu care se stabileşte la fel ca şi domeniul oricărei variabile.

Un nume de clasă poate fi redeclarat ca orice nume. În acest caz, putem folosi numele respectiv într-un domeniu inclus în care este redefinit folosind construcţia:

```
class :: nume_clasa
```

în locul numelui.

Exemplu:

```
class a {
  ...
};
...
{        // instrucţiune compusa aflata in domeniul numelui a
 double a;        // redeclararea lui a
 a = 3.1415;      // atribuire corecta
 class::a x;      // x este o instantiere a lui a
 ...
}
```

Această regulă se utilizează şi în cazul construcţiilor *struct*, *union* şi *enum*. De fapt, construcţiile *struct* şi *union* se consideră că definesc clase ale căror elemente membru sînt toate publice.

Un nume al unui element membru al unei clase care nu este public are un *domeniu de tip clasă*. Aceasta înseamnă că el poate fi folosit numai în corpul funcţiilor membru ale clasei respective.

Elementele membru statice vor fi prefixate de numele clasei urmat de operatorul de rezoluţie (vezi paragraful precedent).

O clasă poate conţine ca şi date membru obiecte care sînt instanţieri ale altei clase, dar nu ale clasei respective.

Exemplu:

```
class clasa {
 ...
 clasa1 obiect1;   // obiect1 este instantiere a clasei clasa1
 clasa obiect;     // eroare; o clasa nu poate contine ca obiecte
                   // membru instantieri ale ei

 ...
};
```

În schimb, se pot defini date membru ale unei clase care să fie pointeri sau referinţe spre obiectele clasei respective.

Exemplu:

```
class clasa {
 ...
 clasa *pobiect;   // pointer spre un obiect de tip clasa
 clasa& robiect;   // referinta la un obiect de tip clasa
 ...
};
```

Clasele pot fi declarate incomplet.

O astfel de declaraţie are formatul:

class *nume*;

Astfel de declaraţii se pot folosi şi în cazul construcţiilor *struct* şi *union*.

Exemplu:

```
class a;
struct s;
union u;
class b {
 ...
 a *pa;
 s *ps;
 u *pu;
 ...
};
```

22.4. Vizibilitate

Un nume este *vizibil* în domeniul său dacă nu este redefinit în blocuri incluse în domeniul respectiv. Un nume redefinit în blocuri din domeniul său, devine temporar ascuns. Un nume cu domeniul de tip fişier poate fi făcut vizibil în domeniul în care este redefinit, folosind operatorul de rezoluţie iar dacă numele respectiv este numele unei clase, atunci el va fi precedat de cuvîntul cheie corespunzător: *class, struct* sau *union*.

Domeniul de vizibilitate al unui nume este acea parte a domeniului său în care el poate fi utilizat.

De obicei, domeniul de vizibilitate al unui nume coincide cu domeniul său.

22.5. Durata de viaţă a datelor

Durata de existenţă a datelor este legată de clasa de memorie a acestora. Prin *durata de viaţă* a datelor se înţelege perioada în care ele sînt alocate în memorie.

Există 3 feluri de durată: *statică, locală* şi *dinamică*.

Durata statică - Înseamnă că data respectivă este alocată în memorie pe perioada execuţiei programului: de la lansare şi pînă la terminarea execuţiei programului.

Datele care au un domeniu de tip fişier sînt date cu durată statică.

Datele care au un domeniu de tip fişier sînt date globale sau locale fişierului dacă ele sînt declarate cu ajutorul cuvîntului cheie *static*. În ambele cazuri, ele au o *durată statică*. De asemenea, datele care au un domeniu de tip local au o durată statică, dacă sînt declarate cu ajutorul cuvîntului cheie *static*.

În concluzie, datele globale, precum şi cele declarate cu ajutorul cuvîntului *static* au o durată statică.

Durata locală - Este durata datelor *automatice*. Acestea sînt date cu domeniu de tip local şi care sînt alocate, la execuţie, pe stivă sau în regiştri. Ele nu conţin, în declaraţia lor, cuvîntul *static*. Alocarea pe stivă se face cînd controlul programului ajunge la blocul în care sînt declarate. Cînd controlul programului iese din blocul respectiv, datele se elimină de pe stivă.

Durata dinamică - Este durata datelor alocate în memoria *heap*. Acestea se alocă şi se eliberează la execuţie prin funcţii sau operatori corespunzători. Ea se realizează de către programator. În acest scop, în limbajele C şi C++, se pot utiliza funcţiile *malloc* şi *free*.

De obicei, în limbajul C++ se utilizează operatorii *new* şi *delete* (vezi capitolul 20).

22.6. Alocarea şi dezalocarea obiectelor

Alocarea obiectelor se face în funcţie de durata de viaţă a obiectelor. Obiectele de durată statică şi locală se alocă automat.

Obiectele dinamice se alocă de către programator. În acest scop, de obicei, se utilizează operatorul *new*.

În mod analog, *dezalocarea* obiectelor se realizează automat dacă ele au durată statică sau locală şi de către programator dacă sînt dinamice. Obiectele alocate cu ajutorul operatorului *new* se dezalocă cu ajutorul operatorului *delete*.

Alocarea obiectelor se mai numeşte şi *crearea* sau *construirea* obiectelor. Dezalocarea obiectelor se mai numeşte şi *distrugerea* obiectelor.

Alocarea obiectelor statice care sînt globale se realizează înainte de execuţia funcţiei *main* a programului. Ele se dezalocă la terminarea programului ca parte a procedurii de ieşire din funcţia *main*.

Obiectele locale se alocă în momentul în care domeniul lor devine activ, adică atunci cînd controlul programului ajunge la instanţierea lor. Ele se dezalocă în momentul în care controlul programului iese din domeniul lor.

Alocarea obiectelor este o operaţie mai complexă decît alocarea datelor de tip predefinit sau definit de utilizator. Alocarea datelor de tip predefinit sau definit de utilizator se poate face împreună cu iniţializarea datelor respective. În cazul obiectelor, iniţializarea datelor membru este o problemă mai complexă din cauza protecţiei datelor respective. De aceea, se pune problema ca iniţializarea obiectelor să fie o operaţie care se realizează la alocarea lor, prin funcţii membru speciale. Aceste funcţii membru, care realizează alocarea şi iniţializarea obiectelor, se consideră că ele construiesc obiectul care se instanţiază. De aceea, ele se numesc *constructori*.

Un constructor este o funcţie membru al unei clase care se apelează la fiecare instanţiere. El are acelaşi nume ca şi numele clasei.

Dezalocarea unui obiect este şi el un proces complex care se realizează cu o funcţie membru specială numită *destructor*. Destructorul unui obiect se apelează la distrugerea obiectului. El se apelează automat sau uneori explicit de către programator. Apelul explicit al destructorului se face pentru a distruge obiectele dinamice. Pentru celelalte obiecte, destructorul se apelează automat la încetarea existenţei lor:

- la ieşirea prin funcţia *exit* pentru obiectele globale;
- la ieşirea din domeniul unui obiect de durată locală.

22.7. Iniţializare

Datele pot fi iniţializate prin declaraţiile (definiţiile) lor.

În general, datele de durată statică (globale sau care au clasa de memorie *static*, adică declaraţia lor începe prin cuvîntul cheie *static*)(vezi cap. 5 şi 6) *neiniţializate*, au valoarea iniţială egală cu zero. Celelalte categorii de date

dacă nu sînt inițializate, au o valoare inițială *nedefinită*.

Datele de tipuri predefinite și cele definite de utilizator se inițializează folosind formatele din limbajul C, care au fost precizate în capitolul 6 și paragraful 10.1.

În limbajul C++ se pot inițializa și datele de tip utilizator definite cu ajutorul construcției *union*. Aceasta este posibil numai pentru prima componentă a reuniunii.

Elementele unui tablou, unei structuri, unei reuniuni sau un enumerator, se pot inițializa prin *expresii constante*, adică expresii care conțin operanzi ce pot fi evaluați la compilare, la întîlnirea lor. În aceste expresii se poate folosi operatorul *sizeof*.

Datele care nu sînt tablouri, structuri, reuniuni sau de tip enumerare, se pot inițializa prin expresii care nu este necesar să fie expresii constante. În acest caz, operanzii expresiilor care se utilizează la inițializare, trebuie să poată fi evaluați în momentul în care controlul programului ajunge la ei.

Exemplu:

```
int f(int n)
{
 int i = n+10;
 /* parametrul n are precizata valoarea cind controlul programului
    ajunge la evaluarea expresiei n+10 si anume valoarea lui este egala
    cu a parametrului efectiv de la apelul lui f */
 int j = k-2   // eroare; k nu este definit la intilnirea expresiei k-2
 int k=3;
...
}
```

Un loc aparte îl ocupă inițializarea obiectelor. Așa cum s-a spus mai sus, obiectele se inițializează cu ajutorul constructorilor care se apelează în mod automat la instanțierea lor.

Datele membru ale obiectelor statice (*obiecte de durată statică*) care nu sînt inițializate au valoarea inițială zero.

Datele membru ale celorlalte obiecte (*obiecte de durată locală* sau *dinamică*) care nu sînt inițializate, au o valoarea inițială nedefinită.

Datele membru statice se inițializează în afara constructorilor. Aceasta, deoarece o dată membru statică este o zonă comună care nu se multiplică la fiecare instanțiere a clasei respective.

Inițializarea unei astfel de date se realizează la fel ca o dată globală obișnuită, adică printr-o definiție a datei respective în care este prezentă și

valoarea de iniţializare, definiţie care se scrie în afara corpului oricărei funcţii.

Exemplu:

```
class dc {
     int zi,luna,an;
     static int zz,ll,aa;
  public:
     ...
};
```

Datele membru statice *zz, ll, aa* se vor iniţializa astfel:

int dc :: zz = 1;
int dc :: ll = 1;
int dc :: aa = 1600;

Se observă prezenţa numelui clasei şi a operatorului de rezoluţie pentru a specifica faptul că *zz, ll* şi *aa* sînt date membru ale clasei *dc*. În absenţa numelui clasei şi a operatorului de rezoluţie, datele *zz, ll, aa* devin date globale iniţializate cu valorile respective.

Atribuiri de forma:

```
dc :: zz = 1; dc :: ll = 1; dc :: a = 1600;
```

sînt posibile, dar numai dacă sînt scrise în corpul unei funcţii membru (datele respective sînt protejate). Astfel de atribuiri, de obicei, nu sînt considerate a fi iniţializări. Ele se realizează numai dacă funcţia membru care le conţine este apelată.

Dacă datele membru statice nu se iniţializează, ele nu trebuie definite în modul indicat mai sus. Ele sînt alocate în mod automat şi au valoarea iniţială egală cu zero.

22.8. Constructor

Datele de tip predefinit sau definit de utilizator se alocă în mod automat, în conformitate cu declaraţia sau definiţia acestora. Odată cu alocarea datelor se pot face şi iniţializări.

În cazul obiectelor, acestea se alocă la instanţierea lor. De asemenea, obiectele pot fi iniţializate la instanţiere. În acest scop, utilizatorul poate defini constructori, care sînt funcţii membru de acelaşi nume cu numele clasei. Se pot defini mai mulţi constructori pentru o clasă. În acest caz, ei sînt funcţii supraîncărcate şi deci ei diferă prin numărul şi/sau tipurile parametrilor.

Valorile de iniţializare se transferă constructorului şi ele joacă acelaşi rol ca parametrii efectivi de la apelurile funcţiilor obişnuite. Ele formează o listă care se include între paranteze rotunde şi sînt prezente în declaraţia (definiţia) obiectelor. În felul acesta, o declaraţie sau definiţie de obiect poate avea formatul:

nume_clasa nume_obiect(*lista*);

unde:

lista - Este formată dintr-o expresie sau mai multe, separate prin virgule.

Lista, împreună cu parantezele care o includ, sînt absente dacă obiectul nu se iniţializează sau dacă există un constructor cu toţi parametri impliciţi.

Exemplu:

Se consideră clasa *complex* definită ca mai jos:

```
class complex {
    double real;
    double imag;
  public:
    complex(double x=0,double y=0)

    /* constructor pentru numere complexe; implicit se instantiaza
       numarul complex cu ambele parti egale cu zero */
    {
        real = x;
        imag = y;
    }
  ...
};
```

Exemple de instanţieri ale clasei *complex*:

```
complex z;            // se instantiaza numarul complex
                      // initializat implicit: z = 0+0*i
complex r(3);         // se instantiaza numarul complex
                      // r = 3+0*i
complex i(0,1);       // se instantiaza numarul complex
                      // i = 0+1*i
complex c(1.5,-1.5);  // se instantiaza numarul complex
                      // c = 1,5-1,5*i
```

La iniţializare, se utilizează regulile de la apelurile funcţiilor supraîncărcate dacă există mai mulţi constructori. Dacă există un singur

constructor, atunci se aplică regula de la apelurile funcţiilor din limbajul C, adică parametrii efectivi (în cazul de faţă expresiile prin care se face iniţializarea) se convertesc spre tipurile parametrilor formali corespunzători ai constructorului.

Această regulă se aplică şi în cazul clasei *complex*, de mai sus, care are un singur constructor. Mai jos, dăm un exemplu de clasă cu mai mulţi constructori:

Exemplu:

```
class dc {
    int zi,luna,an;
  public:
    dc()   // constructor fara parametri
    {
        zi = 1;
        luna = 1;
        an = 1600;
    }

    dc(int z,int l,int a=1995)
    /* constructor cu trei parametri de tip int */
    {
        zi = z;
        luna = l;
        an = a;
    }

    dc(int z,char *denl, int a);
    /*constructor pentru initializare cu denumirea
      lunii calendaristice */
        ...
    };
```

Exemple de instanţieri ale clasei *dc*:

```
dc d1;   // se apeleaza constructorul fara parametri
         // d1: zi=1, luna=1, an=1600
dc d2(15,9,1995);   // se apeleaza constructorul cu toti
                    // parametri de tip int
                    // d2: zi=15, luna=9, an=1995
dc d3(15,9);   // se apeleaza acelasi constructor ca la d2
               // si se obtine acelasi rezultat
dc d4(15,"septembrie",1997);
/* se apeleaza constructorul cu parametrul char *denl */
```

```
dc *pd = new dc(15,9);
/* se apeleaza constructorul ca si in cazul obiectelor d2 si d3;
   obiectul se aloca in memoria heap, datele membru se initializeaza
   ca la obiectele d2 si d3; pd are ca valoare adresa de inceput a
   obiectului alocat in memoria heap */
```

În acest exemplu s-a definit un constructor fără parametri. Un astfel de constructor se numeşte *constructor implicit.*

În cazul în care există un constructor implicit nu se mai poate defini, pentru clasa respectivă, un constructor cu toţi parametrii impliciţi. Într-adevăr, un astfel de constructor conduce la ambiguitate la instanţierea obiectelor. De exemplu, dacă alături de constructorul *dc()* al *clasei dc*, am defini constructorul:

```
dc(int z=1,int l=1,int a=1600)
```

atunci instanţierea:

```
dc d;
```

este ambiguă, deoarece se pot apela ambii constructori.

Prezenţa constructorilor nu este obligatorie. Se pot defini clase şi fără constructori. În acest caz, compilatorul C++ generează în mod automat un constructor fără parametri, adică un constructor implicit. Acesta are rol numai de alocare a obiectelor clasei respective. Constructorii definiţi de programator sînt necesari numai în cazul în care se doreşte iniţializarea obiectelor la instanţierea lor.

În cazul în care o clasă are cel puţin un constructor şi nici unul nu este constructorul implicit, atunci nu se pot instanţia obiecte neiniţializate. Aceasta, deoarece compilatorul nu creează constructorul implicit pentru clasele care au cel puţin un constructor.

Exemplu:

```
class complex {
     double real;
     double imag;
 public:
     complex(double x,double y)
     {
        real = x;
        imag = y;
     }
  ...
};
```

În acest caz, se pot instanţia numai obiecte cu ambele părţi iniţializate:

```
complex z(1,2);
complex z1;     //eroare: nu exista constructor implicit
```

Declaraţia (definiţia) obiectelor pentru care lista de iniţializare se reduce la un singur parametru, poate fi scrisă într-un format care să nu difere de cel utilizat la iniţializarea variabilelor simple. Astfel:

nume_clasa nume_obiect = expresie;

realizează instanţierea obiectului *nume_obiect* al clasei *nume_clasa*, instanţiere la care se apelează un constructor pentru care primul parametru are ca valoare, valoarea lui *expresie*. Alţi parametri, sau nu există la constructorul apelat sau sînt parametri impliciţi.

Exemplu:

```
class complex {
     double real;
     double imag;
 public:
     complex(double x=0,double y=0)
     {
         real = x;
         imag = y;
     }
  ...
};
```

Exemple de instanţieri:

```
complex z;          //z = 0+0*i
complex z1(1);      //z1 = 1+0*i
complex z2 = 1;     //z2 = 1+0*i
complex z3(1,2);    //z3 = 1+2*i
```

În cazul în care dorim să instanţiem obiecte atît iniţializate, cît şi neiniţializate, putem folosi un constructor implicit vid, care se va apela la instanţierea obiectelor neiniţializate.

Exemplu:

```
class dc {
     int zi,luna,an;
 public:
     dc()   //constructor implicit vid
       {    //se utilizeaza pentru instantierea
       }    //obiectelor neinitializate
```

```
    dc(int z,int l,int a)
     /* constructor utilizat pentru initializarea obiectelor */
    {
        zi = z;
        luna = l;
        an = a;
    }
    ...
};
```

Se pot utiliza instanţieri de forma:

```
dc d;    // se apeleaza constructorul implicit
dc d1(1,2,1997);    // se instantiaza obiectul d1 initializat
                    // astfel: zi=1, luna=2, an=1997
```

Parametrii unui constructor pot fi de orice tip, cu excepţia tipului definit de clasa pentru care este funcţie membru. Deci, dacă *clasa* este numele unei clase, atunci nu se poate defini un constructor de antet:

```
clasa(clasa p)
```

În schimb, constructorul unei clase poate avea ca parametri pointeri sau referinţe la obiectele clasei respective.

Deci:

```
clasa(clasa *p)
```

şi

```
clasa(clasa& p)
```

sînt antete corecte de constructori.

Constructorul:

```
clasa(const clasa& p)
```

este un constructor special care permite copierea obiectelor. El se numeşte *constructor de copiere*. Constructorul de copiere poate avea şi alţi parametri care însă trebuie să fie impliciţi.

Constructorul de copiere se apelează într-o instanţiere de felul celei de mai jos:

```
clasa c(...);     // instantiere cu initializare obisnuita
clasa c1 = c;     // se apeleaza constructorul de copiere;
                  // c1 este o copie a lui c
```

Dacă programatorul nu defineşte un constructor de copiere, atunci

compilatorul generează un constructor de copiere implicit, dacă este nevoie de el.

Exemplu:

Mai jos, definim un constructor de copiere pentru numerele complexe.

```
class complex {
     double real;
     double imag;
 public:
     complex(double x=0,double y=0)
     {
         real = x;
         imag = y;
     }
     complex(const complex& c)  // constructor de copiere
     {
         real = c.real;
         imag = c.imag;
     }
     ...
};
```

Exemple de instanţieri:

```
complex z(1,2);     // z = 1+2*i
complex z1 =z;      // se apeleaza constructorul de copiere
                    // z1 = 1+2*i
complex z2(z);      // se apeleaza constructorul de copiere
                    // z2 = 1+2*i
```

Datele membru ale unei clase pot fi date arbitrare dar nu obiecte ale clasei respective. În particular, datele membru pot fi obiecte ale unei alte clase.

Un exemplu simplu este cel oferit de primitivele care se utilizează la definirea figurilor pe ecranul setat în mod grafic.

Considerăm clasa *Punct* care defineşte un punct pe ecran. Acesta are o poziţie definită prin coordonatele sale (coloana şi linia în care se afişează punctul respectiv). Definim clasa *Punct* ca mai jos:

```
class Punct {
     int x;  // coloana
     int y;  // linia
```

```
public:
    Punct(int col=0,int linia = 0)
        // constructor: punctul implicit este cel de
        // coordonate (0,0)
    {
        x = col;
        y = linia;
    }
    ...
};
```

Un dreptunghi se poate trasa dacă se definesc două vîrfuri diagonal opuse.

De obicei, se consideră vîrful din stînga sus al dreptunghiului şi cel din dreapta jos. Un vîrf al dreptunghiului este un obiect al clasei Punct. De aceea, putem defini clasa *Dreptunghi* cu ajutorul a două obiecte ale clasei *Punct*:

```
class Dreptunghi {
 Punct st_sus;
 Punct dr_jos;
 ...
};
```

Constructorul clasei *Dreptunghi* trebuie să transfere valori pentru parametrii constructorului clasei *Punct*.

Acest lucru se realizează modificînd antetul constructorului, ca mai jos.

Fie clasa *cl* definită astfel:

```
class cl {
 cl1 c1;
 cl2 c2;
 ...
 cln cn;
 ...
};
```

unde:

cl1,cl2,...,cln - Sînt nume de clase, în prealabil definite, care nu neapărat sînt toate distincte. Ele, au fiecare, constructori pentru iniţializarea obiectelor.

Constructorul *cl* transferă valorile de iniţializare pentru obiectele membru *c1,c2,...,cn* prin antetul său de format:

```
c1(...): c1(...),c2(...),...,cn(...)
```

În acest antet vor lipsi obiectele pentru care nu se transferă date de inițializare.

Reluînd exemplul de mai sus, completăm definiția clasei *Dreptunghi* cu doi constructori:

```
class Dreptunghi {
    Punct st_sus;
    Punct dr_jos;
  public:
    Dreptunghi(int dr,int j): dr_jos(dr,j)
      //se instantiaza dreptunghiul cu virfurile:
      //      stinga sus:  (0,0)    - implicit
      //      dreapta jos: (dr,j)
    {
    }
    Dreptunghi(int st,int sus,int dr,int jos):
            st_sus(st,sus),dr_jos(dr,jos)
      // se instantiaza dreptunghiul cu virfurile:
      //      stinga sus:  (st,sus)
      //      dreapta jos: (dr,jos)
    {
    }
  ...
};
```

Metoda de inițializare a obiectelor, care sînt date membru ale unei clase, poate fi utilizată și pentru date membru care nu sînt obiecte.

Astfel, un constructor de forma:

```
complex(double x,double y)
{
 real = x;
 imag = y;
}
```

poate fi scris și sub forma:

```
complex(double x,double y):real(x),imag(y)
{
}
```

În încheierea acestui paragraf, enumerăm caracteristicile de bază ale constructorilor.

Constructorii nu returnează nici o valoare la revenirea din ei. Mai mult decît atît, antetul lor constituie o excepţie, deoarece nu este admis cuvîntul cheie *void*, cuvînt care trebuie să fie prezent în antetul funcţiilor care nu returnează o valoare.

În cazul în care o clasă are obiecte membru care au constructori, aceştia se apelează înainte de a se apela constructorul clasei respective.

Obiectele claselor care au cel puţin un constructor nu pot fi componente ale unei reuniuni.

Spre deosebire de funcţiile obişnuite, adresa constructorului nu se poate determina.

De obicei, constructorii sînt funcţii membru publice, dar nu se pot apela explicit la fel ca celelalte funcţii membru.

Un constructor poate fi apelat explicit pentru situaţii de felul celui de mai jos:

```
complex z = complex(1,2);
```

În acest caz a fost apelat constructorul clasei *complex* pentru a crea un obiect anonim care este iniţializat cu valorile 1 pentru partea reală şi 2 pentru partea imaginară. Apoi, obiectul respectiv este copiat în obiectul *z*.

22.9. Destructor

Destructorii sînt funcţii care pot fi consideraţi că acţionează în sens invers faţă de constructori. Ei au multe caracteristici în comun cu constructorii, dar între ei există şi diferenţe.

Numele unui destructor este numele clasei precedat de caracterul "~". Un destructor nu are parametri şi el este unic pentru o clasă. Dacă programatorul nu a definit un destructor, atunci compilatorul generează un destructor pentru clasa respectivă.

Antetul destructorilor nu conţine cuvîntul *void*, deşi ei nu returnează valori. De aceea, destructorii au antetul:

~*nume_clasa*()

în interiorul definiţiei clasei *nume_clasa*.

În afara definiţiei clasei, antetul destructorului va fi:

nume_clasa :: ~*nume_clasa*()

Ca şi în cazul constructorilor, adresa destructorului nu poate fi determinată.

Obiectele unei clase care au un destructor şi/sau cel puţin un constructor nu se pot utiliza ca şi componente ale unei reuniuni.

Dacă la instanţierea unui obiect s-au apelat mai mulţi constructori, atunci la distrugerea lui, destructorii se vor apela în ordine inversă.

Pentru un obiect global, destructorul se apelează ca parte a procedurii *exit* de la terminarea execuţiei funcţiei *main*. De aceea, un astfel de destructor nu trebuie să apeleze funcţia *exit* deoarece se intră într-un ciclu infinit.

Pentru un obiect local,destructorul se apelează cînd controlul programului iese din domeniul lui (se iese din blocul în care este declarat).

Obiectele dinamice nu pot fi distruse automat. Distrugerea se realizează de către programator deoarece numai el ştie cînd un astfel de obiect nu mai este necesar.

Destructorul poate fi apelat de programator atît direct, cît şi indirect prin intermediul operatorului *delete*.

Noi utilizăm operatorul *delete* pentru a distruge obiecte create dinamic cu ajutorul operatorului *new* (obiecte dinamice).

Exemple:

1. Fie tipul *complex* definit în exemplele precedente.

```
...
complex *pz;
...
pz = new complex(1,2);
// in memoria heap se construieste un obiect de tip
// complex cu partea reala egala cu 1 si partea imaginara
// egala cu 2; pz pointeaza spre obiectul respectiv
...
delete pz;    // se distruge obiectul creat mai sus prin new
```

2. Se defineşte clasa *string* ca mai jos:

```
class string {
        char *sir;
        int lung;
    public:
        string(char *);
        ~string();
        ...
};
```

```
string :: string(char *s)   // definitia constructorului
{
    lung = strlen(s) + 1;
    sir = new char[lung];
    strcpy(sir,s);
}
string :: ~string()   // definitia destructorului
{
    delete sir;
}
```

Exemple de instanţieri:

```
string sir_de_caractere("Acesta este un sir\
                         de caractere");
// se creeaza obiectul sir_de_caractere in memoria heap si
// se initializeaza cu textul :
//      "Acesta este un sir de caractere"
...
delete sir_de_caractere;   // se distruge obiectul apelindu-se
                           // indirect destructorul
```

Apelul direct al destructorului se poate face numai dacă numele lui este precedat de numele clasei care este urmat de operatorul de rezoluţie:

```
...
string s("exemplu");
...
s.string :: ~string();   // apel direct al destructorului
```

Exerciţii:

22.1 Să se definească tipul abstract de date *complex* care să aibă funcţii membru pentru modul, argument, pentru accesul la partea reală şi imaginară a obiectelor de tip complex, pentru afişarea obiectelor complexe, precum şi constructori pentru iniţializare şi copiere.

FIŞIERUL BXXII1.H

```
class complex {
    // date membru protejate(private)
    double real;   // partea reala
    double imag;   // partea imaginara
  public:   // functii membru neprotejate
    complex(double x=0,double y=0);
   // constructor folosit la initializare
```

```
    complex(const complex&);
     // constructor de copiere

    double modul();         // modulul numarului complex
    double arg();           // argumentul numarului complex
    double retreal();       // returneaza partea reala
    double retimag();       // returneaza partea imaginara
    void afiscomplex();  // afiseaza numarul complex
};    // sfirsit definitie clasa complex
```

FIŞIERUL BXXII1

```
#ifndef __BXXII1_H
#include "BXXII1.H"
#define __BXXII1_H
#endif
#ifndef __MATH_H
#include <math.h>
#define __MATH_H
#endif
#ifndef __PI
#define  PI 3.14159265358979
#define __PI
#endif

// constructor pentru initializarea obiectelor
inline complex :: complex(double x,double y)
{
 real = x;
 imag = y;
}

// constructor de copiere
inline complex :: complex(const complex& z)
{
 real = z.real;
 imag = z.imag;
}

inline double complex :: modul()
// calculeaza modulul numarului complex
{
 return sqrt(real*real+imag*imag);
```

```
}

double complex :: arg()
// calculeaza argumentul numarului complex
{
 double a;

 if(real==0 && imag==0)
    return 0.0;
 if(imag==0)
    if(real > 0)
       return 0.0;
    else
       return PI;
 if(real==0)
    if(imag > 0)
       return PI/2;
    else
       return (3*PI)/2;

 // real != 0 si imag != 0
 a = atan(imag/real);
 if(real < 0)   // real < 0 si y !=0
    return a+PI;
 else      // real > 0
    if(imag < 0)   // real > 0 si imag < 0
       return 2*PI+a;

 // real > 0 si imag > 0
 return a;
}  // sfirsit arg

inline double complex :: retreal()
// returneaza partea reala a numarului complex
{
 return real;
}

inline double complex :: retimag()
// returneaza partea imaginara a numarului complex
{
 return imag;
}
```

```
inline void complex :: afiscomplex()
//afiseaza numarul complex
{
 printf("%g+i*(%g)\n",real,imag);
}
```

22.2 Să se scrie un program care realizează următoarele:

- citeşte perechi de numere care reprezintă, fiecare, partea reală şi respectiv partea imaginară a unui număr complex;
- afişează:
  - numărul complex citit;
  - rădăcina pătrată din fiecare număr complex citit;
  - suma numerelor complexe citite.

PROGRAMUL BXXII2

```
#include <stdio.h>
#include <conio.h>
#include "BXXII1.CPP"

main()
/* - citeste perechi de numere care reprezinta partea reala si respectiv cea
   imaginara a unui numar complex;
  - afiseaza:
   - numarul complex citit;
   - radacina patrata din numarul complex citit;
   - suma numerelor complexe citite.
*/
{
 double x;
 double y;
 double sx = 0;
 double sy = 0;

 do {
    printf("partea reala =");
    if(scanf("%lf",&x) != 1)
        break;   //nu mai sint numere
    printf("partea imaginara =");
    if(scanf("%lf",&y) != 1){   //eroare
        printf("partea imaginara eronata\n");
        printf("se reia citirea numarului\n");
```

```
        fflush(stdin); //videaza zona tampon de la
                       //intrarea standard
        continue;      //se reia ciclul de citire
      }

      //se insumeaza partea reala si cea imaginara a
      //numerelor complexe citite
      sx += x;
      sy += y;

      complex z(x,y); //se construieste numarul complex avind
                      //partile citite; se apeleaza constructorul
                      //care realizeaza initializarea obiectului
                      //z: z=x+i*y
      z.afiscomplex(); //afiseaza numarul citit
      double m = z.modul(); //calculeaza modulul lui z
      double a=z.arg();     //calculeaza argumentul
                            //numarului complex
      double m1 = sqrt(m);  //radacina patrata din modulul z
      double a1 = a/2;      //semiargumentul numarului z

      //rz1 = sqrt(z)
      printf("modulul = %g\targumentul = %g\n",m,a);
      double preal = m1*cos(a1);
      double pimag = m1*sin(a1);
      complex rz1 = complex(preal,pimag);

      //rz2 = sqrt(z);
      complex rz2 = complex(-preal,-pimag);
      printf("radacina patrata\n");
      printf("sqrt(z)1 = ");
      rz1.afiscomplex();
      printf("sqrt(z)2 = ");
      rz2.afiscomplex();
    } while(1);
    complex sz = complex(sx,sy);    //sz = suma
    printf("suma numerelor complexe citite = ");
    sz.afiscomplex();  //afiseaza suma numerelor complexe citite
  }
```

22.3 Să se extindă tipul abstract complex, definit în exerciţiul 22.1, aşa încît să se poată realiza următoarele operaţii asupra obiectelor complexe:

- adunare;
- scădere;
- negativare;
- înmulţire;
- împărţire;
- citirea de la intrarea standard a componentelor unui obiect complex.

FIŞIERUL BXXII3.H

```
enum Boolean {false,true};
class complex {
    // date membru protejate (private)
    double real;
    double imag;
 public:        // functii membru neprotejate
    complex(double x=0,double y=0); // constructor
    complex(const complex&); // constructor de copiere
    double modul();   // modulul numarului complex
    double arg();   // argumentul numarului complex
    double retreal();   // returneaza partea reala
    double retimag();   // returneaza partea imaginara
    void afiscomplex(); // afiseaza numarul complex

    // functii membru noi

    Boolean citcomplex();
    // citeste componentele numarului complex;
    // returneaza false la EOF

    void adcomplex(complex *z1,complex *z2);
    // calculeaza z = z1+z2
    void sccomplex(complex *z1,complex *z2);
    // calculeaza z = z1-z2
    void negcomplex(complex *z1);
    // calculeaza z = -z1
    void mulcomplex(complex *z1,complex *z2);
    // calculeaza z = z1*z2
    Boolean divcomplex(complex *z1,complex *z2);
    // calculeaza z = z1/z2; returneaza false la
```

```
    // impartirea cu zero
};
```

Fişierul de mai jos, conţine definiţiile funcţiilor membru noi.

Definiţiile funcţiilor membru vechi se preiau din fişierul BXXII1.CPP.

FIŞIERUL BXXII3

```
#ifndef __BXXII3_H
#include "BXXII3.H"
#define __BXXII3_H
#endif
#define __BXXII1_H
#ifndef __BX18_CPP
#include "BX18.CPP"
#define __BX18_CPP
#endif
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif

// se defineste BXXII1_H pentru a nu mai include fisierul
// BXXII1.H prin includerea fisierului BXXII1.CPP

#include "BXXII1.CPP" // se preiau definitiile functiilor
                      // membru vechi

// definitiile functiilor membru noi

Boolean complex :: citcomplex()
/* - citeste componentele numarului complex;
   - returneaza:
        false la sfirsitul fisierului;
        true altfel.
*/
{
 double preal;
 double pimag;

 if(pcit_double("Partea reala:",&preal) == 0)
    return false;  //EOF
 if(pcit_double("Partea imaginara: ",&pimag) == 0)
```

```
    return false;  // EOF
  real = preal;
  imag = pimag;
  return true;
 }

inline void complex :: adcomplex(complex *z1,
                                  complex *z2)
// calculeaza z = z1+z2
{
 real = z1 -> real+z2 -> real;
 imag = z1 -> imag+z2 -> imag;
}

inline void complex :: sccomplex(complex *z1,
                                  complex *z2)
// calculeaza z = z1-z2
{
 real = z1 -> real - z2 -> real;
 imag = z1 -> imag - z2 -> imag;
}

inline void complex :: negcomplex(complex *z1)
// calculeaza z = -z1
{
 real = -z1 -> real;
 imag = -z1 -> imag;
}

inline void complex :: mulcomplex(complex *z1,
                                   complex *z2)
// calculeaza z = z1*z2
{
 real = z1 -> real*z2 -> real - z1 -> imag*z2
        -> imag;
 imag = z1 -> real*z2 -> imag + z1 -> imag*z2
        -> real;
}

Boolean complex :: divcomplex(complex *z1,
                               complex *z2)
// calculeaza z = z1/z2
{
```

```
   double d = z2 -> real *z2 -> real + z2 -> imag*z2
              -> imag;
   if(d == 0)   //divizor nul
      return false;
   real = (z1 -> real*z2 -> real + z1 -> imag*z2
            -> imag)/d;
   imag = (z1 -> imag*z2 -> real - z1 -> real*z2
            -> imag)/d;
   return true;
  }
```

22.4 Să se scrie un program care citeşte numerele complexe *a*, *b*, *c*, rezolvă şi afişează rădăcinile ecuaţiei de gradul 2:

a*x*x+b*x+c=0

PROGRAMUL BXXII4

```
#include <stdlib.h>
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#include "BXXII3.CPP"

main()
/* - citeste: a,b,c
   - rezolva ecuatia: a*x*x+b*x+c=0
   - afiseaza solutiile ecuatiei
*/
{
 complex a,b,c;
 char er[] = "s-a tastat EOF\n";

 if(a.citcomplex() == false){
    printf(er);
    exit(1);
 }
 if(b.citcomplex() == false){
    printf(er);
    exit(1);
 }
 if(c.citcomplex() == false){
    printf(er);
    exit(1);
```

```
}
if(a.retreal() == 0 && a.retimag() == 0 &&
   b.retreal() == 0 && b.retimag() == 0 &&
   c.retreal() == 0 && c.retimag() == 0) {
   printf("ecuatie nedeterminata\n");
   exit(0);
}
if(a.retreal() == 0 && a.retimag() == 0 &&
   b.retreal() == 0 && b.retimag() == 0) {
   printf("ecuatia nu are solutie\n");
   exit(1);
}
if(a.retreal() == 0 && a.retimag() == 0){
   printf ("ecuatie de gradul 1\n");

   complex x,z;

      z.divcomplex(&c,&b);  // z = c/b
      x.negcomplex(&z);     // x = -z
   printf("x=");
   x.afiscomplex();
   exit(0);
} // obiectele x si z se distrug in acest punct

//ecuatie de gradul 2

complex bp;

   bp.mulcomplex(&b,&b); // bp = b*b

complex patru(4,0);
complex patrua;

   patrua.mulcomplex(&a,&patru);  // 4*a

complex patruac;

   patruac.mulcomplex(&patrua,&c);  // 4*a*c

complex deltap;

   deltap.sccomplex(&bp,&patruac);  // b*b-4*a*c
```

```
double r = deltap.modul();
    r = sqrt(r);
double argument = deltap.arg();
    argument = argument/2;

complex delta(r*cos(argument),r*sin(argument));
 // delta = sqrt(b*b-4*a*c)

complex mb;

    mb.negcomplex(&b);   // mb = -b

complex doia;

    doia.adcomplex(&a,&a);   // doia = a+a

complex x;

    x.adcomplex(&mb,&delta);  // x = -b+delta

complex x1;

    x1.divcomplex(&x,&doia);  // x1 = (-b+delta)/(2*a)
    printf("x1 = ");
    x1.afiscomplex();   // afiseaza radacina x1

complex x2;

    x.sccomplex(&mb,&delta); // x = -b-delta
    x2.divcomplex(&x,&doia);  // x2 = (-b-delta)/(2*a)
    printf("x2 = ");
    x2.afiscomplex();   // afiseaza radacina x2
}
```

22.5 Să se definească tipul abstract *dc* pentru implementarea datei calendaristice. În acest scop se defineşte clasa *dc* care are următoarele componente de tip *int*:

- *zi*, *luna*, *an*;
- *minzz*, *minll*, *minaa* - pentru data calendaristică considerată ca dată validă minimă;
- *maxzz*, *maxll*, *maxaa* - pentru data calendaristică considerată ca

dată validă maximă;

- *tnrz* - este un tablou ale cărui elemente definesc numărul de zile ale lunilor calendaristice (luna februarie se consideră că are 28 de zile).

Componentele *minzz, minll, minaa, maxzz, maxll, maxaa* şi *tnrz* nu se multiplică la fiecare instanţiere a clasei *dc*, ele fiind utilizate în comun de către funcţiile membru. De aceea, ele se definesc ca date membru statice.

În mod implicit, data calendaristică minimă se consideră 1 ianuarie 1600, iar cea maximă se consideră 31 decembrie 4900.

Clasa *dc* are 3 funcţii pentru verificarea corectitudinii datelor calendaristice. Una este o funcţie membru obişnuită care verifică data calendaristică a obiectului curent, iar celelalte două verifică corectitudinea datelor calendaristice definite de datele membru statice *minzz, minll, minaa, maxzz, maxll,* şi *maxaa*. Aceste două funcţii sînt funcţii membru statice. Aceste 3 funcţii apelează o funcţie membru statică *v_calend* care validează o dată calendaristică definită prin parametrii ei. Faptul că o dată calendaristică este dintr-un an bisect sau nu, se determină folosind trei funcţii, una pentru obiectele de tip *dc* şi două pentru cele două date membru statice. Aceste funcţii folosesc în comun o funcţie *bisect* care stabileşte dacă data calendaristică definită de parametri ei este o dată dintr-un an bisect sau nu.

Funcţia *bisect* este o funcţie obişnuită. Ea se defineşte în acelaşi fişier cu funcţiile membru ale clasei.

Clasa are un constructor implicit care se utilizează la instanţierea obiectelor fără iniţializarea datei calendaristice nestatice (zi, luna, şi an).

Amintim că datele statice se iniţializează independent, în afara constructorilor. Constructorii clasei *dc* vor verifica corectitudinea datelor calendaristice statice.

În caz de eroare, se afişează un mesaj corespunzător şi se forţează datele implicite.

Pentru instanţierea obiectelor iniţializate se utilizează un constructor cu trei parametri pentru cele 3 date membru care nu sînt statice (zi, luna şi an).

Constructorul utilizat în acest scop, verifică atît datele statice, cît şi cele care se iniţializează prin constructorul respectiv.

În caz de eroare, se afişează un mesaj corespunzător şi se instanţiază un obiect cu data calendaristică minimă.

Alte funcţii membru:

- constructor de copiere;

- *retzi*: returnează ziua din data obiectului curent;
- *retluna*: returnează luna din data obiectului curent;
- *retan*: returnează anul din data obiectului curent;
- *afisdata*: afişează data calendaristică a obiectului curent;
- *datamin*: returnează data minimă;
- *datamax*: returnează data maximă;
- *modifmin*: modifică data minimă;
- *modifmax*: modifică data maximă;
- *citdata*: citeşte o dată calendaristică;
- *adzi*: adună un număr de zile la data obiectului curent;
- *difdata*: determină diferenţa, în zile, dintre două date calendaristice;
- *zi_din_an*: returnează ziua din an pentru data obiectului curent;
- *ziua_si_luna*: determină luna şi ziua din lună din parametrii an şi ziua din an;
- *nr_zile_luna*: returnează numărul de zile din luna calendaristică a obiectului curent;
- *verif_min_max*: verifică datele minimă şi maximă; dacă o dată este eronată, se dă un mesaj de eroare şi se forţează data implicită: 1 ianuarie 1600 pentru data minimă şi 31 decembrie 4900 pentru data maximă.

FIŞIERUL BXXII5.H

```
#ifndef __Boolean
#define __Boolean
enum Boolean {false,true};
#endif

class dc {
// date membru protejate

    int zi,luna,an;
    static int minzz,minll,minaa;
```

```
    static int maxzz,maxll,maxaa;
    static int tnrz[13];
    static char *tdenluna[13];
    static char *ermin;
    static char *ermax;
    static char *erdc;
    static void afiser(char *sir);

    // afiseaza textul spre care pointeaza sir

    static void verif_min_max();

    // verifica datele minima si maxima; la eroare se da mesaj
    // si se forteaza data implicita corespunzatoare

  public:
    Boolean valid_dc();
    /* verifica corectitudinea datei calendaristice a
       obiectului curent; la eroare returneaza false */

    static Boolean valid_dc_min();
    /* verifica corectitudinea datei minime; la eroare
       returneaza false */

    static Boolean valid_dc_max();
    /* verifica corectitudinea datei maxime; la eroare
       returneaza false */

    Boolean bisect_dc();
    /* returneaza:
          true  - daca data obiectului curent este dintr-un an bisect;
          false - altfel.
    */

    static Boolean bisect_dc_min();
    /* returneaza:
          true  - daca data minima este dintr-un an bisect;
          false - altfel
    */

    static Boolean bisect_dc_max();
    /* returneaza:
          true  - daca data maxima este dintr-un an bisect;
```

```
        false - altfel.
   */

static Boolean v_calend(int z,int l,int a);
 /* returneaza:
        true  - daca data definita de parametrii z, l si a este valida;
        false - altfel.
  */

dc();   // constructor implicit pentru obiecte neinitializate

dc(int z,int l,int a);
// constructor pentru initializarea obiectelor

dc(const dc&); // constructor pentru copiere

int retzi();      // returneaza ziua din data obiectului curent

int retluna(); // returneaza luna din data obiectului curent

int retan();      // returneaza anul din data obiectului curent

void afisdata();    // afiseaza data obiectului curent

dc *datamin(); // returneaza un pointer spre data minima

dc *datamax(); // returneaza un pointer spre data maxima

void modifmin(); /* schimba data calendaristica minima cu
                    data obiectului curent */

void modifmax(); /* schimba data calendaristica maxima cu
                    data obiectului curent */

int citdata();      /* citeste o data calendaristica;
                       returneaza 0 la sfirsit de fisier
                       si 1 altfel */

Boolean adzi(long z); /* aduna, la data obiectului curent,
                         numarul de zile dat de
                        parametrul z */
```

```
long difdata(dc *d);
/* returneaza diferenta, in numar de zile, dintre data
   calendaristica a obiectului curent si cea spre care pointeaza d */

char *denluna();   /* returneaza un pointer spre denumirea
                      lunii datei obiectului curent */

int zi_din_an();   /* returneaza ziua din an pentru data
                      obiectului curent */

Boolean ziua_si_luna(int z,int a);
/* determina data calendaristica a obiectului curent din
     z - ziua din an
   si din
     a - anul calendaristic */

int nr_zile_luna(); /* returneaza numarul de zile din
                       luna calendaristica a datei obiec-
                       tului curent */
}; // sfirsit definitia clasei dc
```

Definiţiile funcţiilor membru şi iniţializarea datelor statice se dau într-un fişier de extensie .CPP

FIŞIERUL BXXII5

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __Boolean
enum Boolean {false,true};
#define __Boolean
#endif
#ifndef __DC_H
#include "BXXII5.H" // definitia clasei dc
#define __DC_H
#endif

// initializarea datelor statice

int dc::minzz = 1;          // implicit, data calendaristica
int dc::minll = 1;          // minima este
```

```
int dc::minaa = 1600;   // 1 ianuarie 1600
int dc::maxzz = 31;     // implicit, data calendaristica
int dc::maxll = 12;     // maxima este
int dc::maxaa = 4900;   // 31 decembrie 4900

// initializarea tabloului cu numarul zilelor din lunile calendaristice
int dc::tnrz[13] =
    {0,31,28,31,30,31,30,31,31,30,31,30,31};

// initializarea textelor de eroare
char *dc::erdc = "data calendaristica eronata\n";
char *dc::ermin = "data minima eronata\n";
char *dc::ermax = "data maxima eronata\n";

// initializarea tabloului cu denumirile lunilor calendaristice
char *dc::tdenluna[13] = {
    "luna ilegala",
    "ianuarie",
    "februarie",
    "martie",
    "aprilie",
    "mai",
    "iunie",
    "iulie",
    "august",
    "septembrie",
    "octombrie",
    "noiembrie",
    "decembrie"
};

// functie obisnuita folosita de functiile membru

inline int bisect(int a)
/* returneaza:
        1 daca a defineste un an bisect;
       0 altfel.
*/
{
 return a%4==0 && a%100 || a%400 == 0;
}
```

// functii membru

```
Boolean dc::v_calend(int z,int l,int a)
/* returneaza:
        true daca data calendaristica definita de parametrii z, l si a este
                valida;
        false in caz contrar. */
{
 if(a < 1600 || a > 4900)
    return false;
 if(a < dc::minaa || a > dc::maxaa)
    return false;
 if(l < 1 || l > 12)
    return false;
 if(a==dc::minaa && l<dc::minll ||
   a==dc::maxaa && l>dc::maxll)
    return false;
 if(z < 1 || z > tnrz[l]+(l==2 && bisect(a)))
    return false;
 if(a==dc::minaa && l==dc::minll &&
    z < dc::minzz || a==dc::maxaa &&
    l==dc::maxll && z > dc::maxzz)
    return false;
 return true;
}

inline void dc::afiser(char *sir)
// afiseaza textul spre care pointeaza sir
{
 printf("%s\n",sir);
}

inline Boolean dc::valid_dc()
/* - verifica corectitudinea datei calendaristice a obiectului curent;
   - la eroare returneaza false. */
{
 return v_calend(zi,luna,an);
}

inline Boolean dc::valid_dc_min()
/* - verifica corectitudinea datei minime;
   - la eroare returneaza false. */
{
```

```
 return v_calend(dc::minzz,dc::minll,dc::minaa);
}

inline Boolean dc::valid_dc_max()
/* - verifica corectitudinea datei maxime;
   - la eroare returneaza false. */
{
 return v_calend(dc::maxzz,dc::maxll,dc::maxaa);
}

Boolean dc::bisect_dc()
/* returneaza true daca anul calendaristic al obiectului curent
   este bisect */
{
 if(bisect(an))
     return true;
 else
     return false;
}

Boolean dc::bisect_dc_min()
/* returneaza true daca anul minim este bisect */
{
 if(bisect(dc::minaa))
     return true;
 else
     return false;
}

Boolean dc::bisect_dc_max()
/* returneaza true daca anul maxim este bisect */
{
 if(bisect(dc::maxaa))
     return true;
 else
     return false;
}

void dc::verif_min_max()
/* - verifica datele minima si maxima;
    - daca o data este eronata se da un mesaj de eroare si se forteaza
      data implicita corespunzatoare. */
{
```

```
if(dc::valid_dc_min() == false){
    dc::afiser(dc::ermin); //se forteaza data minima
                           //implicita
    dc::minzz = 1;
    dc::minll = 1;
    dc::minaa = 1600;
  }
  if(dc::valid_dc_max() == false){
    dc::afiser(dc::ermax); //se forteaza data maxima
                           //implicita
    dc::maxzz = 31;
    dc::maxll = 12;
    dc::maxaa = 4900;
  }
}

dc::dc(int z,int l,int a)
/* - constructor pentru initializarea obiectelor;
   - se verifica data minima, maxima si cea care se instantiaza. */
{
 dc::verif_min_max();

 // initializarea obiectului care se instantiaza

 zi = z;
 luna = l;
 an = a;
 if(valid_dc()==false){
    afiser(dc::erdc);
    zi = dc::minzz;
    luna = dc::minll;
    an = dc::minaa;
 }
}

inline dc::dc() /* constructor pentru obiecte neinitializate */
{
 dc::verif_min_max();
}

dc::dc(const dc &d) /* constructor de copiere */
{
 zi = d.zi;
```

```
 luna = d.luna;
 an = d.an;
 if(valid_dc() == false){
     afiser(dc::erdc);
     zi = dc::minzz;
     luna = dc::minll;
     an = dc::minaa;
 }
}

inline int dc::retzi()    //returneaza ziua obiectului curent
{
 return zi;
}

inline int dc::retluna() //returneaza luna obiectului curent
{
 return luna;
}

inline int dc::retan()     //returneaza anul obiectului curent
{
 return an;
}

inline void dc::afisdata() //afiseaza data obiectului curent
{
 printf("zi: %d\tluna: %d\tan: %d\n", zi,luna,an);
}

dc *dc::datamin()     //returneaza un pointer spre data minima
{
 dc *d_min = new dc(dc::minzz,dc::minll,dc::minaa);
 return d_min;
}

dc *dc::datamax()   //returneaza un pointer spre data maxima
{
 dc *d_max = new dc(dc::maxzz,dc::maxll,dc::maxaa);
 return d_max;
}
```

```
void dc::modifmin()
// schimba data calendaristica minima cu data obiectului curent
{
 dc::minzz = zi;
 dc::minll = luna;
 dc::minaa = an;
 dc::verif_min_max();
}

void dc::modifmax()
// schimba data calendaristica maxima cu data obiectului curent
{
 dc::maxzz = zi;
 dc::maxll = luna;
 dc::maxaa = an;
 dc::verif_min_max();
}

int dc::citdata()
// citeste o data calendaristica;
// returneaza:
//     0 la sfirsit de fisier;
//     1 altfel.
{
 int i,c;

 for(;;){
    for(;;){
       printf("ziua: ");
       if((c=scanf("%d",&i)) == 1 && i > 0 &&
           i <= 31)
          break;
       if(c == EOF)
          return 0;
       printf("ziua eronata\n");
       fflush(stdin);
    }
    zi=i;
    for(;;){
       printf("luna: ");
       if((c=scanf("%d",&i)) == 1 && i > 0 &&
           i <= 12)
          break;
```

```
        if(c == EOF)
          return 0;
        printf("luna eronata\n");
        fflush(stdin);
    }
    luna=i;
    for(;;){
        printf("anul: ");
        if((c=scanf("%d",&i)) == 1 && i >= 1600 &&
             i <= 4900)
          break;
        if(c == EOF)
          return 0;
        printf("anul eronat\n");
        fflush(stdin);
    }
    an=i;
    if(valid_dc() == true)
        break;
    printf("data calendaristica: %d /%d/%d\
             eronata\n", zi,luna,an);
 }
 return 1;
}

int dc::zi_din_an()
// returneaza ziua din an pentru data obiectului curent
{
 int b = bisect(an);
 int z=zi;

 for(int i=1;i < luna;i++)
    z +=dc::tnrz[i]+(i==2&&b);
 return z;
}

Boolean dc::ziua_si_luna(int z,int a)
// determina data calendaristica din:
//      z - ziua din an
// si din
//     a - anul
{
 int b = bisect(a);
```

```
  if(z > 365+b){  // ziua din an eronata
     printf("ziua = %d eronata\n",z);
     return false;
  }
  if(a < 1600 || a > 4900)  // anul eronat
  {
     printf("anul = %d eronat\n",a);
     return false;
  }
  int i = 1;
  do{
     int j = dc::tnrz[i]+(i==2 && b);
     if(z <= j)
        break;
     z -= j;
     i++;
  }while(1);
  zi = z;
  luna = i;
  an = a;
  return valid_dc();
}

inline char *dc::denluna()
// returneaza un pointer spre denumirea lunii obiectului curent
{
 return luna<1 || luna>12 ? dc::tdenluna[0] :
     dc::tdenluna[luna];
}

Boolean dc:: adzi(long z)
/* aduna la data obiectului curent numarul de zile dat de parametrul z */
{
 long totzile = zi_din_an();
 totzile += z; /* numarul total de zile care se considera dupa
                  31 decembrie din anul precedent celui curent */

 int ziledinan = 365 + bisect(an);
 if(totzile >= 0)
    while(totzile >= ziledinan) {  /* se socotesc anii */
        totzile -= ziledinan;
        an++;
        if( a <= 4900)
```

```
            ziledinan = 365+bisect(an);
      }
   else{
      an--;
      ziledinan = 365 +bisect(an);
      while(-totzile >= ziledinan){ /* se socotesc anii */
         totzile += ziledinan;
         an--;
         if(an >= 1600)
            ziledinan = 365+bisect(an);
      }
      totzile += ziledinan; /* determina numarul de zile
                               din anul determinat */
   }
   if(totzile == 0){
      zi = 31;
      luna = 12;
      an--;
      return valid_dc();
   }
   return ziua_si_luna(totzile,an);
}

long dc::difdata(dc *d)
/* returneaza diferenta, in numar de zile, dintre data
   calendaristica curenta si cea spre care pointeaza d */
{
   long zidinan1 = zi_din_an(); /* numarul de zile din
                                   anul curent */
   long zidinan2 = d -> zi_din_an();/* numarul de zile din
                                       anul definit de poin-
                                       terul d */
   long difzi = zidinan1 - zidinan2;
   int an1,an2;
   if(an < d -> an) {
      an1 = an;
      an2 = d -> an;
   }
   else{
      an1 = d -> an;
      an2 = an;
   }
```

```
    long difzian = 0;
    while(an1 < an2){
       int b = bisect(an1);
       difzian += 365+b;
       an1++;
    }
    if(an < d -> an)
       difzian = -difzian;
    return difzi+ difzian;
}
```

Observaţii:

1. Funcţiile membru statice, de obicei, se apelează calificînd numele lor cu numele clasei urmat de operatorul de rezoluţie. Mai sus a fost apelată funcţia membru statică *v_calend* fără a respecta acest lucru. Aceasta este posibil dacă nu există definită în acelaşi program încă o altă funcţie cu acelaşi nume şi listă a parametrilor formali.

   Pentru a evita astfel de situaţii se recomandă să se respecte regula cu privire la calificarea numelor funcţiilor membru statice prin numele clasei urmat de operatorul de rezoluţie.

2. Funcţiile membru care nu sînt statice se apelează numai în legătură cu un obiect al clasei pentru care ele sînt funcţii membru. Apelurile unei astfel de funcţii au forma:

   obiect.nume_functie(...)

   sau

   pobiect -> nume_functie(...)

   unde:

   pobiect - este un pointer spre tipul implementat prin clasa pentru care *nume_functie* este funcţie membru.

Mai sus, s-au apelat funcţii membru nestatice ale clasei *dc* în corpul altor funcţii membru nestatice ale aceleeaşi clase fără a respecta regula de mai sus. De exemplu,în corpul funcţiei membru *citdata* se apelează funcţia membru *valid_dc* astfel:

```
if(valid_dc() == true)
 break;
```

Acesta este corect deoarece în mod implicit apelul de mai sus este realizat folosind pointerul *this*:

```
if(this -> valid_dc() == true)
```

```
break;
```

22.6 Să se scrie un program care citeşte, de la intrarea standard, o succesiune de date calendaristice şi afişează fiecare dată calendaristică împreună cu cea a zilei următoare.

PROGRAMUL BXXII6

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#include "BXXII5.CPP"

main()
/* citeste o succesiune de date calendaristice si afiseaza datele citite
   impreuna cu data zilei urmatoare */
{
  for(;;) {
      dc data_citita;
      if(data_citita.citdata() == 0)
          break;   //s-a intilnit EOF
      data_citita.afisdata();   // afiseaza data citita
      dc data_urm = data_citita;   // copiere
      data_urm.adzi(1L);   // determina data zilei urmatoare
      data_urm.afisdata();   // afiseaza data zilei urmatoare
  }
}
```

22.7 Să se scrie un program care realizează următoarele:

– citeşte date calendaristice care sînt situate între două date limită;

– afişează fiecare dată citită împreună cu diferenţa, în număr de zile, dintre data citită şi datele limită.

Cele două date limită se definesc prin argumente în linia de comandă a programului. Data minimă este prima dată calendaristică din linia de comandă.

PROGRAMUL BXXII7

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#include "bxxii5.cpp"

main(int argc,char *argv[])
/* - citeste date calendaristice situate intre doua date limita;
   - afiseaza datele citite impreuna cu diferenta, in numar de
     zile, dintre datele limita si data citita
*/
{
 if(argc != 7){
    printf("numar argumente = %d eronat\n",argc);
    exit(1);
 }
 int tabarg[6];

 for(int i=1;i < 7;i++){
    char *p = argv[i];
    for(int j=0;*p;j++,p++)
        if(*p < '0' || *p > '9'){
           printf("argument eronat:\
                  %c\t%d\n",*p,*p);
           exit(1);
        }
        if(j > 4){
           printf("argument eronat: %s\n",argv[i]);
           exit(1);
        }
    tabarg[i-1] = atoi(argv[i]); // conversia
                                 // argumentului
 }
```

```
    dc liminf(tabarg[0],tabarg[1],tabarg[2]);
    dc limsup(tabarg[3],tabarg[4],tabarg[5]);

    liminf.afisdata();     // afiseaza limita inferioara
    limsup.afisdata();     // afiseaza limita superioara
    liminf.modifmin();     // defineste data statica minima
    limsup.modifmax();     // defineste data statica superioara

    // citirea datelor calendaristice

    for(;;){
       dc data_crt;
       if(data_crt.citdata() == 0)
          break;   //s-a intilnit EOF
       data_crt.afisdata();   // afiseaza data citita
       long difi,difs;
       difi=liminf.difdata(&data_crt);
       difs = limsup.difdata(&data_crt);
       printf("diferenta in numar de zile =\
              %ld\t%ld\n",difi,difs);
       dc *pinf;
       pinf = pinf -> datamin();
       dc *psup ;
       psup = psup -> datamax();
       printf("limita inferioara\n");
       pinf -> afisdata();
       printf("limita superioara\n");
       psup -> afisdata();
       delete pinf;
       delete psup;
    }
}
```

22.8 Să se definească tipul abstract *punct* care să se utilizeze la instanţierea punctelor din plan prin coordonate rectangulare.

FIŞIERUL BXXII8.H

```
class punct {
    double x;
    double y;
 public:
    punct(double abs=0,double ord=0);
    punct(const punct&);
```

```
    int citpunct();     /* citeste coordonatele punctului;
                           returneaza:
                                  0 la EOF;
                                  1 altfel.       */
    void afispunct(); /* afiseaza coordonatele punctului */
    void xtrans(double dx); /* translatie pe directia
                               abscisei */
    void ytrans(double dy); /* translatie pe directia
                               ordonatei */
    double retx();     // returneaza abscisa
    double rety();     // returneaza ordonata
};
```

Funcţiile membru se definesc separat într-un fişier de tip CPP.

FIŞIERUL BXXII8

```
#ifndef __PUNCT_H
#include "BXXII8.H"
#define __PUNCT_H
#endif

inline punct::punct(double abs,double ord)
/* - constructor pentru instantierea obiectelor de tip punct;
   - implicit se instantiaza originea axelor. */
{
 x = abs;
 y = ord;
}

inline punct::punct(const punct& p)
// constructor de copiere
{
 x = p.x;
 y = p.y;
}

int punct::citpunct()
/* - citeste coordonatele punctului;
   - returneaza:
          0 - la EOF;
          1 - altfel.
*/
{
```

```
int c;

for(;;){
   printf("Abscisa=");
   if((c=scanf("%lf",&x))==1)
      break;
   if(c==EOF)
      return 0;
   printf("nu s-a tastat un numar\n");
   fflush(stdin);
}
for(;;){
   printf("Ordonata=");
   if((c=scanf("%lf",&y))==1)
      return 1;
   if(c==EOF)
      return 0;
   printf("nu s-a tastat un numar\n");
   fflush(stdin);
}
}

inline void punct::afispunct()
// afiseaza coordonatele punctului
{
 printf("x=%g\ty=%g\n",x,y);
}

inline void punct::xtrans(double dx)
// translatie in directia axei x
{
 x += dx;
}

inline void punct::ytrans(double dy)
// translatie in directia axei y
{
 y += dy;
}

inline double punct::retx() // returneaza abscisa
{
 return x;
```

```
}

inline double punct::rety() // returneaza ordonata
{
 return y;
}
```

22.9 Să se scrie un program care generează obiecte de tip *punct* ale căror coordonate se generează aleator. Programul afişează punctele care aparţin ecranului, iar în final se indică:

- numărul m al punctelor afişate;
- numărul n al punctelor generate;
- raportul m/n.

PROGRAMUL BXXII9

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#include "BXXII8.CPP"

main()
/* - genereaza puncte cu coordonate aleatoare;
   - afiseaza punctele care apartin ecranului (80 coloane
     a 25 linii);
   - afiseaza numarul m al punctelor afisate;
   - afiseaza numarul n al tuturor punctelor generate;
   - raportul m/n.
*/
{
 int m=0,n=0;

 for(;;){
    double xaleat = rand()%100;
```

```
        double yaleat = rand()%100;
        punct pct(xaleat,yaleat);
        n++;
        if(xaleat > 0 && xaleat <= 80 &&
           yaleat > 0 && yaleat <= 25){   // afiseaza punctul
           m++;
           pct.afispunct();
           if(m%22 == 0){
              printf("Actionati o tasta pentru\
                      a continua\n");
              printf("Actionati zero pentru\
                      a termina\n");
              if(getch() == '0')
                 break;
           }
        }
   }
   printf("numarul punctelor afisate = %d\n",m);
   printf("numarul punctelor generate = %d\n",n);
   printf("raportul m/n = %g\n", (double) m/n);
}
```

22.10 Să se definească tipul abstract *vector*.

Un vector este definit de două puncte. Unul este originea vectorului, iar celălalt este vîrful lui.

FIŞIERUL BXXII10.H

```
#ifndef __PUNCT_H
#include "BXXII8.CPP"
#define __PUNCT_H
#endif

class vector {
     punct origine;
     punct virf;
  public:  // constructori
     vector(double xo,double yo,double xv,
            double yv);
     vector(double xv,double yv);
     vector(const vector&);
     double modul();   // modulul vectorului
```

```
    double arg();   // argumentul vectorului (unghiul facut
                    // cu axa Ox)

    void x_trans(double dx);
    // translatie in directia axei Ox

    void y_trans(double dy);
    // translatie in directia axei Oy

    punct retorig();   // returneaza originea vectorului
    punct retvirf();   // returneaza virful vectorului

    double prodscalar(vector *v);
    // returneaza produsul scalar dintre vectorul curent si
    // cel spre care pointeaza v
};
```

Observaţie:

Modulul şi argumentul vectorilor se calculează asemănător cu modulul şi argumentul numerelor complexe. În acest caz se face diferenţa dintre coordonatele extremităţilor vectorului.

Funcţiile membru ale clasei *vector* se definesc în fişierul de mai jos, de extensie CPP.

FIŞIERUL BXXII10

```
#ifndef __VECTOR_H
#include "BXXII10.H"
#define __VECTOR_H
#endif
#ifndef __MATH_H
#include <math.h>
#define __MATH_H
#endif
#ifndef __PI
#define PI 3.14159265358979
#define __PI
#endif
```

```
inline vector::vector(double xv,
                      double yv):virf(xv,yv)
// originea are coordonatele egale cu zero (valori implicite)
{
}

inline vector::vector(double xo,double yo,
                       double xv,double yv):
   origine(xo,yo),virf(xv,yv)
// constructor pentru initializarea originii si virfului unui vector
{
}

vector::vector(const vector& v):
   origine(v.origine),virf(v.virf)
// constructor de copiere
{
}

double vector::modul()  // returneaza modulul unui vector
{
 double a = virf.retx() - origine.retx();
 double b = virf.rety() - origine.rety();
 return sqrt(a*a+b*b);
}

double vector::arg()  // returneaza argumentul unui vector
{
 double a = virf.retx() - origine.retx();
 double b = virf.rety() - origine.rety();
 if(a == 0 &&b == 0)
    return 0.0;
 if(b == 0)
    if(a > 0)
       return 0.0;
    else
       return PI;
 if(a == 0)
    if(b > 0)
       return PI/2;
    else
       return(3*PI)/2;
```

```
//a != 0 si b != 0
double c = atan(b/a);
if(a < 0)
    return PI+c; //a < 0 si b != 0
if(b < 0)
    return 2*PI+c; //a > 0 si b < 0

//a > 0 si b > 0
return c;
}

inline void vector::x_trans(double dx)
// translatie in directia axei Ox
{
 origine.xtrans(dx);
 virf.xtrans(dx);
}

inline void vector::y_trans(double dy)
// translatie in directia axei Oy
{
 origine.ytrans(dy);
 virf.ytrans(dy);
}

inline punct vector::retorig()
// returneaza originea vectorului
{
 return origine;
}

inline punct vector::retvirf()
// returneaza virful vectorului
{
 return virf;
}

double vector::prodscalar(vector *v)
/* returneaza produsul scalar dintre vectorul curent si cel spre care
   pointeaza v */
{
 double ax = v -> virf.retx() - v ->
             origine.retx();
```

```
double ay = v -> virf.rety() - v ->
           origine.rety();
double bx = virf.retx() - origine.retx();
double by = virf.rety() - origine.rety();
return ax*bx+ay*by;
}
```

22.11 Să se scrie un program care citeşte coordonatele a doi vectori din plan şi afişează modulul şi argumentul fiecărui vector, precum şi produsul lor scalar.

PROGRAMUL BXXII11

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#include "BXXII10.CPP"

main()
/* - citeste coordonatele a doi vectori;
   - calculeaza si afiseaza:
   - modulul si argumentul fiecarui vector;
   - produsul scalar al celor doi vectori.
*/
{
 punct ov1;  //origine
 punct vv1;  //virf
 punct ov2;  //origine
 punct vv2;  //virf

// citeste primul vector
 ov1.citpunct();
 vv1.citpunct();

// citeste al doilea vector
 ov2.citpunct();
 vv2.citpunct();
```

```
// construieste vectorii
vector v1(ov1.retx(),ov1.rety(),
          vv1.retx(),vv1.rety());
vector v2(ov2.retx(),ov2.rety(),
          vv2.retx(),vv2.rety());

// afiseaza modulul si argumentul lui v1 si v2
printf("Vectorul v1\n");
printf("modul = %g\t argument = %g\n",
      v1.modul(),v1.arg());
printf("Vectorul v2\n");
printf("modul = %g\t argument = %g\n",
     v2.modul(),v2.arg());

// afiseaza produsul scalar
printf("(v1,v2) = %g\n",v1.prodscalar(&v2));
}
```

22.12 Să se definească tipul abstract *sir* pentru instanţierea şirurilor de caractere. Acest tip este definit în cele mai multe lucrări relative la limbajul C++.

Interesul manifestat pentru acest tip rezultă din faptul că şirurile de caractere sînt utilizate frecvent în aproape toate programele. Mai jos, prezentăm o implementare a tipului *sir* analogă cu cea indicată în lucrarea [7].

Ca date membru alegem un pointer spre zona de memorie în care se păstrează caracterele şirului şi lungimea acestuia în număr de caractere (fără caracterul *nul* de la sfîrşitul şirului).

FIŞIERUL BXXII12.H

```
class sir {
    char *psir;
    int lung;
 public:
    sir(char *s); /* constructor pentru initializarea
                     obiectului cu pointerul spre sirul
                     de caractere; acesta se pastreaza
                     in memoria heap */

    sir(int nrcar=70); /* constructor care rezerva zona
                          de memorie in memoria heap si
                          pastreaza in ea sirul vid */
```

```
    sir(const sir&);  // constructor de copiere
    ~sir();           // destructor
    int retlung();    // returneaza lungimea sirului
    void afsir();     // afiseaza sirul de caractere
    int citsir();     // citeste un sir de caractere

    Boolean atribsir(sir *s);
    /* - transfera sirul spre care pointeaza s in zona rezervata pentru
         sirul obiectului curent;
       - daca nu exista zona suficienta, se truncheaza sirul spre care
         pointeaza s si se returneaza valoarea false;
       - altfel se returneaza true. */
};
```

Mai jos, se definesc funcţiile membru ale clasei *sir*.

FIŞIERUL BXXII12

```
#ifndef __Boolean
#define __Boolean
enum Boolean {false,true};
#endif
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __STRING_H
#include <string.h>
#define __STRING_H
#endif
#ifndef __SIR_H
#include "BXXII12.H"
#define __SIR_H
#endif

sir::sir(char *s)
/* constructor utilizat la initializarea obiectului cu pointerul spre copia
   in memoria heap a sirului spre care pointeaza s */
{
 lung = strlen(s);  // lungimea sirului
 psir = new char[lung+1];   // rezerva zona in memoria heap
 strcpy(psir,s);  // transfera sirul in memoria heap
}
```

```
sir::sir(int dim)
/* - constructor care rezerva zona pentru siruri de dimensiunea dim;
   - pastreaza sirul vid in zona respectiva. */
{
 lung = dim;                    // determina lungimea
 psir = new char[lung+1];       // rezerva zona
 *psir = '\0';                  // pastreaza sirul vid
}

sir::sir(const sir& s)  // constructor de copiere
{
 lung = s.lung;                 // lungimea sirului
 psir = new char[lung+1];       // rezerva zona
 strcpy(psir,s.psir);           // copiaza sirul
}

inline sir::~sir()
/* destructor: elibereaza zona din memoria heap ocupata de sir */
{
 delete psir;
}

inline int sir::retlung()
// returneaza lungimea sirului: numarul caracterelor sirului fara
// caracterul nul
{
 return lung;
}

inline void sir::afsir()
// afiseaza sirul spre care pointeaza psir
{
 printf(psir);
 printf("\n");
}

int sir::citsir()
/* - citeste un sir de la intrarea standard si-l pastreaza in zona heap
     rezervata pentru obiectul curent;
   - returneaza:
        0 - la sfirsit de fisier
       -1 - la trunchierea sirului citit
        1 - altfel. */
```

```
{
 char temp[255];

 if(gets(temp)==0)
    return 0;          // s-a intilnit sfirsitul de fisier
 strncpy(psir,temp,lung);
 *(psir+lung)='\0';
 if(strlen(temp) > lung)
    return -1;         // trunchiere
 else
    return 1;
}

Boolean sir::atribsir(sir *s)
/* - transfera sirul spre care pointeaza s in zona rezervata pentru sirul
     obiectului curent;
   - daca nu exista zona suficienta, se truncheaza sirul spre care pointea-
     za s si se returneaza valoarea false; altfel se returneaza true */
{
 strncpy(psir,s -> psir,lung);
 if(strlen(s -> psir) > lung)
    return false;
 return true;
}
```

22.13 Să se scrie un program care realizează următoarele operaţii asupra obiectelor de tip *sir*:

- iniţializare;
- citire de şiruri de caractere de la intrarea standard;
- copiere de obiecte de tip *sir*;
- atribuiri de obiecte de tip *sir*;
- afişări de şiruri de caractere din compunerea obiectelor de tip *sir*.

PROGRAMUL BXXII13

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
```

```
#endif
#include "BXXII12.CPP"

main() /* operatii cu obiecte de tip sir */
{
 // instantieri de obiecte de tip sir

 sir sir1("Limbajul C++ este un C mai bun");
 sir sir2("Limbajul C++ suporta stilul de\
          programare:\n\
          - prin abstractizarea datelor;\n\
          - orientata spre obiecte");
 sir sir3;    // instantiere fara initializare; sir3 contine sirul vid
 sir sir4 = sir1;   // instantiere folosind constructorul de
                    // copiere

 // afisarea obiectelor instantiate mai sus
 printf("sir1\n");
 sir1.afsir();
 printf("sir2\n");
 sir2.afsir();
 printf("sir3\n");
 sir3.afsir();
 printf("sir4\n");
 sir4.afsir();

 // citiri de siruri de la intrarea standard
 while(sir3.citsir())  // citirea sirurilor
     sir3.afsir();     // afisarea sirurilor citite

 // atribuiri de siruri
 if(sir3.atribsir(&sir1) == false)
     printf("trunchiere la atribuire\n");
 sir3.afsir();
 if(sir3.atribsir(&sir2) == false)
     printf("trunchiere la atribuire\n");
 sir3.afsir();
}
```

22.10. Funcţie prieten (Friend function)

La începutul capitolului, s-a afirmat că o proprietate de bază a tipurilor abstracte este protecţia datelor membru ale tipului respectiv. Elementele protejate constituie aşa numita implementare a tipului abstract.

Tipurile abstracte se definesc cu ajutorul claselor. Se obişnuieşte să se spună că datele protejate ale unui tip abstract sînt încapsulate în clasa care defineşte tipul respectiv.

Protecţia datelor se realizează prin aceea că la ele au acces numai funcţiile membru ale tipului (clasei). De asemenea, dacă o funcţie membru este protejată, atunci ea poate fi apelată numai prin intermediul unei funcţii membru a tipului (clasei).

Acest mod de lucru, deşi asigură o protecţie bună a elementelor membru protejate ale clasei (elemente protejate prin *private* sau *protected*), uneori este considerat prea rigid. Astfel, deşi există funcţii descrise în C care pot fi utilizate pentru a prelucra instanţieri ale unei clase, ele nu se pot utiliza simplu deoarece nu sînt funcţii membru ale clasei respective şi deci nu au acces la datele membru. Mai mult decît atît, o funcţie membru se apelează totdeauna în dependenţă cu un obiect care este numit *obiectul curent* al apelului. În acest fel, o funcţie membru se apelează prin una din următoarele formate:

nume_obiect.nume_functie_membru(...)

sau

pointer_nume_clasa -> nume_functie_membru()

În cazul funcţiilor obişnuite nu se admit astfel de apeluri, toate datele prelucrate de funcţie se transferă prin parametri sau sînt globale.

De aceea, o funcţie obişnuită poate fi utilizată pentru a prelucra obiectele unei clase dacă ea se modifică în aşa fel ca să devină funcţie membru.

S-a făcut un compromis pentru a admite accesul la elementele protejate şi pentru anumite funcţii care nu sînt membru. Aceste funcţii au fost numite *funcţii prieten* pentru clasa respectivă. Ele trebuie să fie precizate ca atare în definiţia clasei. În acest scop, prototipurile lor sînt prezente în definiţia clasei şi sînt precedate de cuvîntul cheie *friend*.

Exemplu:

Fie tipul definit de utilizator:

```
struct complex {
    double real;
    double imag;
};
```

Funcţia *modul*, pentru calculul modulului unei date de tip *complex* poate fi definită, ca mai jos, ca o funcţie obişnuită:

```
double modul(complex *z)
/* returneaza modulul numarului complex spre care pointeaza z */
{
 return sqrt(z -> real*z -> real +z ->
             imag*z -> imag);
}
```

Dacă se consideră declaraţiile:

```
complex z1 = {1,1};
double d;
```

atunci instrucţiunea de atribuire:

```
d = modul(&z1);
```

atribuie lui *d* modulul lui z1 = 1+i.

Acelaşi lucru se poate realiza implementînd tipul abstract *complex* ca în exerciţiul 22.1.:

```
class complex {
    double real;
    double imag;
 public:
    complex(double x=0,double y=0)
    {
        real = x;
        imag = y;
    }
    double modul()
    {
        return sqrt(real*real+imag*imag);
    }
    ...
};
```

Declaraţia pentru *z1* se scrie:

```
complex z1(1,1);
```

iar instrucţiunea de atribuire devine:

```
d = z1.modul();
```

În acest caz, funcţia *modul* nu are parametru deoarece ea se apelează pentru obiectul *z1*. În corpul funcţiei *modul* este definit pointerul implicit *this* şi acesta are ca valoare chiar adresa lui *z1*.

Expresia:

```
real*real+imag*imag
```

din corpul funcţiei membru *modul*, trebuie considerată ca şi cînd ar fi scrisă sub forma:

```
this -> real*this -> real+this -> imag*this -> imag
```

Funcţia *modul*, definită pentru tipul utilizator *complex* poate înlocui funcţia membru *modul* a clasei *complex* dacă ea se declară ca funcţie prieten a clasei *complex*, ca mai jos:

```
class complex {
    double real;
    double imag;
 public:
    complex(double x=0,double y=0)
    {
        real = x;
        imag = y;
    }
 friend double modul(complex *z);
 ...
};
```

Funcţia *modul*, prieten a clasei *complex*, are aceeaşi definiţie ca şi funcţia *modul* definită pentru tipul *complex* introdus prin construcţia *struct*. De asemenea, ea se apelează ca orice funcţie obişnuită.

În exemplul de faţă, dacă se consideră instanţierea:

```
complex z(1,1);
```

atunci *funcţia prieten modul* se apelează în mod obişnuit, adică prin:

```
d = modul(&z);
```

Spre deosebire de funcţiile membru, în cazul funcţiilor prieten nu mai este definit pointerul implicit *this*. Acest lucru conduce la faptul că o funcţie prieten are un parametru în plus faţă de o funcţie membru care are acelaşi efect.

O funcţie prieten poate fi o funcţie obişnuită (ca în cazul funcţiei *modul* din exemplul de mai sus) sau o funcţie membru a unei alte clase.

Exemplu:

```
class clasa1 {
 ...
 tip functie_membru(...);
 ...
};

class clasa2 {
 ...
 friend tip clasa1::functie_membru(...);
 ...
};
```

Funcţia *functie_membru* este o funcţie membru a clasei *clasa1*. Ea este o funcţie prieten a clasei *clasa2*.

În cazul în care se doreşte ca toate funcţiile membru ale clasei *clasa1* să fie funcţii prieten ale clasei *clasa2*, se poate proceda ca mai jos, în loc de a indica individual fiecare funcţie din clasa *clasa1* că este funcţie prieten:

```
class clasa1; // definitia prescurtata de clasa
class clasa2 {
 ...
 friend clasa1;
 ...
};
```

În acest caz, se spune că *clasa1* este o clasă prieten a clasei *clasa2*.

Proprietatea de clasă prieten *nu* este tranzitivă. Astfel, dacă *clasa1* este o clasă prieten pentru *clasa2* şi *clasa2* este o clasă prieten pentru *clasa3*, aceasta nu implică faptul că *clasa1* este clasă prieten pentru *clasa3*.

Modificatorii de protecţie nu au nici o influenţă asupra unei funcţii prieten. De aceea, specificarea faptului că o funcţie este prieten pentru o clasă, poate fi scrisă în orice punct din interiorul definiţiei clasei respective.

Funcţia prieten nu este protejată şi deci poate fi utilizată fără nici o restricţie ca orice funcţie obişnuită.

Exerciţii:

22.14 Să se scrie o funcţie care ridică un număr complex la o putere întreagă pozitivă.

Un număr complex se poate ridica la o putere întreagă pozitivă folosind formula lui *Moivre*. Aceasta se exprimă prin relaţia:

(r*(cos(a)+i*sin(a)))**n=r**n*(cos(n*a)+i*sin(n*a))

unde:

r - Este modulul numărului complex
a - Este argumentul acestuia.

Tipul *complex* se implementează folosind *clasa* definită în exerciţiul 22.3.

FUNCŢIA BXXII14

```
void cputere(complex& z,int n)
/* ridica la puterea n numarul complex la care z este referinta */
{
 double r;
 double a;

 r = z.modul(); /* - se calculeaza modulul numarului complex la
                     care z este referinta;
                   - se apeleaza functia membru modul a clasei
                     complex */
 a = z.arg(); /* - se calculeaza argumentul numarului complex la
                   care z este referinta;
                 - se apeleaza functia membru arg a clasei com-
                   plex */
 double rlan = pow(r,(double)n);
 double na = n*a;
 z.real = rlan*cos(na);
 z.imag = rlan*sin(na);
}
```

Observaţie:

Funcţia *cputere* nu este o funcţie membru a clasei complex. Ea are acces la componentele *real* şi *imag* ale obiectului de tip complex referit de *z*, numai dacă este o funcţie prieten a clasei *complex*.

22.15 Să se modifice definiţia clasei *complex* din fişierul BXXII3.H inserînd funcţia *cputere* ca funcţie prieten.

FIŞIERUL BXXII15.H

```
enum Boolean {false,true};
```

```
class complex {   // date membru protejate (private)
    double real;
    double imag;
 public:   // functii membru neprotejate
    complex(double x = 0,double y = 0);
    complex(const complex&);
    double modul();
    double arg();
    double retreal();
    double retimag();
    void afiscomplex();
    Boolean citcomplex();
    void adcomplex(complex *z1,complex *z2);
    void sccomplex(complex *z1,complex *z2);
    void negcomplex(complex *z1);
    void mulcomplex(complex *z1,complex *z2);

    Boolean divcomplex(complex *z1,complex *z2);

    // functie prieten
    friend void cputere(complex& z,int n);
    // calculeaza z**n, pentru n natural
};
```

22.16 Să se scrie un program care ridică numărul complex $1+i$ la puterea n. Numărul n este un întreg citit de la intrarea standard.

PROGRAMUL BXXII16

```
#ifndef __PI
#define PI 3.14159265358979323 8
#define __PI
#endif
#ifndef __MATH_H
#include <math.h>
#define __MATH_H
#endif
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif

// pentru a nu include fisierul BXXII3.H se defineste __BXXII3_H
#define __BXXII3_H
```

```
// se include definitia clasei complex din fisierul BXXII15.H
#include "BXXII15.H"

// se includ definitiile functiilor membru
#include "BXXII3.CPP"

// se include definitia functiei prieten cputere
#include "BXXII14.CPP"
#include <stdlib.h>

main()
/* citeste intregul n, calculeaza si afiseaza (1+i)**n */
{
 int n,c;
 complex z(1,1);  //z = 1+i
 complex zlan;    //zlan = 0+0*i

 // citeste pe n
 for(;;){
    printf("exponent=");
    if((c = scanf("%d",&n)) == 1)
       break;
    printf("nu s-a tastat un intreg\n");
    if(c == EOF){
       printf("s-a tastat EOF\n");
       exit(1);
    }
    fflush(stdin);
 }

 complex complex_unu(1,0);
 int m;

 m = n < 0 ? -n : n;   // calculeaza abs(n)
 if(m == 0){           // (1+i)**0 = 1+0.i
    complex_unu.afiscomplex();
    exit(0);
 }
 if(n == 1) {          // (1+i)**1 = 1+i
    z.afiscomplex();
    exit(0);
 }
 if( n == -1){         // (1+i)**(-1) = 1/(1+i)
```

```
        zlan.divcomplex(&complex_unu,&z);
        zlan.afiscomplex();
        exit(0);
    }

    // m > 1
    cputere(z,m);   // z = z**m
    if(n > 1){
        z.afiscomplex();
        exit(0);
    }

    // putere negativa: z**n = 1/z**abs(n)
    zlan.divcomplex(&complex_unu,&z);
    zlan.afiscomplex();
   }
```

23. SUPRAÎNCĂRCAREA OPERATORILOR

Tipurile abstracte de date se definesc în limbajul C++ cu ajutorul claselor. Ar fi ideal ca tipurile abstracte să se comporte ca şi cele predefinite. Aşa cum s-a arătat în capitolul precedent, există o serie de asemănări între tipurile abstracte şi cele predefinite. Aşa de exemplu, datele de tip abstract (obiectele) se declară la fel ca şi cele de tip predefinit. De asemenea, ele pot fi iniţializate la declarare.

Operaţiile care se pot executa asupra obiectelor sînt bine precizate ca şi în cazul tipurilor predefinite. În cazul tipurilor abstracte, operaţiile se definesc cu ajutorul *funcţiilor membru* şi *prieten*.

Exemplu:

Pentru clasa *complex* se pot defini funcţii membru şi prieten pentru a realiza cele 4 operaţii aritmetice asupra obiectelor de tip *complex*.

În definiţia de clasă, de mai jos, se defineşte o funcţie membru pentru adunarea obiectelor de tip *complex* şi o funcţie prieten pentru scăderea obiectelor de tip *complex*.

```
class complex {
    double real;
    double imag;
 public:
    complex(double x=0,double y=0)
     // constructor
    {
        real = x;
        imag = y;
    }

    // functie membru pentru adunarea obiectelor de tip complex
    complex adcomplex(complex& z2);

    // functie prieten pentru scaderea obiectelor de tip complex
    friend complex sccomplex(complex& z1,
                             complex& z2);

    ...
};
```

În continuare, se pot defini funcţiile *adcomplex* şi *sccomplex* ca mai jos.

```
complex complex::adcomplex(complex& z)
/* returneaza suma dintre obiectul complex curent si cel referit de z */
{
 complex ztemp;

 ztemp.real = real + z.real;
 ztemp.imag = imag + z.imag;
 return ztemp;
}

complex sccomplex(complex& z1,complex& z2)
/* returneaza diferenta z1-z2 */
{
 complex ztemp;

 ztemp.real = z1.real - z2.real;
 ztemp.imag = z1.imag - z2.imag;
 return ztemp;
}
```

Dacă *a*, *b*, *c* sînt 3 obiecte de tip complex instanţiate cu ajutorul declaraţiei:

```
complex a,b,c;
```

atunci instrucţiunea:

(1) `c = a.adcomplex(b);`

atribuie lui *c*, de tip *complex*, rezultatul adunării obiectelor complexe *a* şi *b*.

Menţionăm că apelul:

(2) `a.adcomplex(b)`

returnează un obiect de tip *complex* la fel cum o funcţie de antet:

tip nume_f(...)

unde *tip* este un tip predefinit (diferit de cuvîntul *void*) returnează o valoare de tip *tip*.

Obiectul complex returnat prin apelul (2) (adică suma a+b) se atribuie obiectului complex *c* prin instrucţiunea (1).

În general, dacă *ob1* şi *ob2* sînt instanţieri ale clasei *nume_clasa*:

nume_clasa ob1,ob2;

atunci o expresie de forma:

(3) `ob1 = ob2`

este admisă de compilator şi ea atribuie lui *ob1* valoarea obiectului *ob2*. Această atribuire se realizează printr-o copiere bit cu bit a valorilor componentelor lui *ob2* în zonele de memorie alocate componentelor lui *ob1*. O astfel de atribuire nu este totdeauna corectă.

În cazul nostru, ea este suficientă pentru atribuirea obiectelor de tip *complex*.

În mod analog, instrucţiunea:

(4) `c = sccomplex(a,b);`

atribuie lui *c* obiectul *complex* rezultat prin scăderea obiectului *complex b* din obiectul *complex a*.

În felul acesta se pot introduce şi alte operaţii asupra obiectelor de tip *complex* folosind funcţii membru sau prieten.

Mecanismul de protecţie a datelor private asigură ca să nu se poată realiza alte operaţii asupra obiectelor de tip complex decît cele precizate la definiţia clasei *complex*.

Se observă o analogie între tipurile predefinite *int, long, float, double* şi tipul abstract *complex*.

Cu toate acestea, tipul *complex* definit ca mai sus, nu permite utilizarea operatorilor obişnuiţi pentru a exprima operaţii cu obiecte *complexe* ca în cazul operaţiilor cu numere de tip *int, long, float* sau *double*.

Astfel, dacă *i, j, k* sînt declarate prin declaraţia:

```
int i,j,k;
```

atunci se pot scrie expresii de forma:

(5) `k = i+j;`

sau

(6) `k = i-j;`

Expresii de acest fel sînt valabile şi pentru alte tipuri predefinite. De exemplu, dacă *x, y* şi *z* sînt date de tip *double*:

```
double x,y,z;
```

atunci se pot utiliza expresii similare:

(7) `z = x+y;`

sau

(8) `z = x-y;`

Evident, expresiile de forma (5), (6), (7) şi (8) sînt sugestive şi ar fi de dorit ca ele să poată fi extinse şi pentru obiecte de tip complex. Cu alte cuvinte, am dori ca în locul instrucţiunii (1) să putem folosi o instrucţiune de forma:

(9) `c = a+b;`

iar în locul lui (4), instrucţiunea:

(10) `c = a-b;`

Acest lucru este posibil prin mecanismul de *supraîncărcare a operatorilor*.

Operatorii + şi - trebuie supraîncărcaţi pentru a realiza operaţii corespunzătoare cu obiecte de tip complex.

De altfel, ei sînt supraîncărcaţi pentru tipurile predefinite *int, long, float, double, unsigned, unsigned long* şi *long double*, deoarece se admit expresii de forma:

```
p = q+r;
```

sau

```
p = q-r;
```

pentru operanzi de tipurile predefinite, enumerate mai sus. Supraîncărcările operatorilor + şi - pentru operanzi de tipuri predefinite sînt predefinite. De aceea, pentru a putea utiliza aceşti operatori şi cu operanzi de tipuri abstracte, este nevoie ca la definirea tipurilor respective să se supraîncarce operatorii în mod corespunzător. Ca rezultat al supraîncărcării operatorilor se ajunge ca operatorii respectivi să poată fi utilizaţi în expresii obişnuite cu operanzi obiecte de tipuri abstracte.

Limbajul C++ permite supraîncărcarea numai a operatorilor existenţi în limbaj. Dintre aceştia nu pot fi supraîncărcaţi operatorii:

. :: ? şi :

De asemenea, prin supraîncărcarea operatorilor nu se poate schimba *n-aritatea, prioritatea* sau *asociativitatea* operatorilor, acestea fiind elemente predefinite pentru tipuri predefinite şi deci ele se vor menţine şi pentru tipuri abstracte.

Prin *n-aritate* înţelegem că operatorul este *binar* sau *unar*.

Supraîncărcarea operatorilor se realizează cu ajutorul unor funcţii

membru sau prieten speciale. Specificul lor constă în numele acestora. El se compune din cuvîntul cheie *operator* şi unul sau mai multe caractere care definesc operatorul care se supraîncarcă. Între cuvîntul cheie *operator* şi caracterele care definesc operatorul care se supraîncarcă se află cel puţin un spaţiu alb.

În cazul tipului *complex*, vom folosi pentru supraîncărcarea operatorului + funcţia definită mai jos:

```
complex complex::operator + (complex& z)
/* - supraincarca operatorul + pentru obiecte de tip complex;
   - functia returneaza un obiect complex care reprezinta suma dintre
     obiectul complex curent si cel referit de z.
*/
{
 complex temp;

 temp.real = real + z.real;
 temp.imag = imag + z.imag;
 return temp;
}
```

În acest caz, numele funcţiei este:

operator +

În rest, funcţia care supraîncarcă operatorul + este identică cu funcţia membru *adcomplex* definită mai sus.

Efectul schimbării numelui *adcomplex* prin *operator +* este acela al supraîncărcării operatorului + pentru a admite ca operanzi obiecte de tip *complex*. Deci, o expresie de forma:

```
a+b
```

este legală şi ea are ca rezultat obiectul *complex* rezultat prin adunarea obiectelor *complexe a* şi *b*. Cu alte cuvinte, această expresie poate fi utilizată în locul expresiei:

```
a.adcomplex(b)
```

al cărui sens este mai puţin evident. În felul acesta, instrucţiunea (1) de mai sus, se transcrie sub forma:

```
c = a+b;
```

care este la fel de simplă şi clară ca şi instrucţiunile (5) şi (7) utilizate pentru operanzi de tipurile predefinite *int* şi respectiv *double*.

În mod analog, operatorul - poate fi supraîncărcat schimbînd numele

funcţiei prieten *sccomplex* cu *operator* - ca mai jos:

```
complex operator - (complex& z1,complex& z2)
/* - supraincarca operatorul - pentru obiecte de tip complex;
   - functia returneaza un obiect complex care reprezinta diferenta
   dintre obiectele z1 si z2 */
{
 complex ztemp;

 ztemp.real = z1.real - z2.real;
 ztemp.imag = z1.imag - z2.imag;
 return ztemp;
}
```

În continuare, se pot utiliza expresii de forma:

```
a-b
```

în locul apelului funcţiei *sccomplex*:

```
sccomplex(a,b)
```

Din cele de mai sus se observă că operatorii pot fi supraîncărcaţi prin funcţii membru sau prin funcţii prieten. O diferenţă între cele două tipuri de funcţii constă în numărul de parametri. Astfel, funcţiile membru care supraîncarcă operatorii unari nu au parametri, iar cele care supraîncarcă operatori binari au un singur parametru.

În cazul funcţiilor prieten, numărul parametrilor este egal cu n-aritatea operatorului care se supraîncarcă.

Motivul pentru a utiliza o funcţie membru sau prieten pentru a supraîncărca un operator va rezulta dintr-un paragraf ulterior.

La supraîncărcarea operatorilor pentru obiecte de tip abstract nu se poate face diferenţă între formele prefixate şi postfixate.

Anumiţi operatori prezintă unele particularităţi la supraîncărcarea lor şi de aceea ei vor fi trataţi în paragrafe separate.

În principiu, funcţiile membru care supraîncarcă un operator nu sînt statice.

O excepţie de la această regulă o constituie supraîncărcarea operatorilor unari *new* şi *delete*. Aceştia se pot supraîncărca numai prin funcţii membru statice.

Alţi operatori care prezintă particularităţi la supraîncărcare sînt parantezele, operatorul -> şi operatorul de atribuire (=).

Mai sus, am considerat supraîncărcarea operatorilor binari pentru cazul cînd ambii operanzi sînt de tipul abstract *complex*.

Se pune problema de a utiliza operatorii respectivi şi cu operanzi diferiţi, adică în expresii de forma:

a+k, k+a, a-k, k-a

unde:

k - Este o variabilă de tip *int*, iar *a* un obiect de tip complex.

Există două soluţii pentru a putea utiliza expresii de această formă. O primă soluţie este supraîncărcarea operatorului + nu numai pentru operanzi de tip *complex*, ci şi pentru operanzi de tipuri diferite.

A doua soluţie constă în conversia operandului care nu este de tip *complex* într-un obiect de tip *complex* înainte de a aplica operatorul *plus* supraîncărcat pentru operanzi de tip *complex*.

Acest lucru se poate realiza destul de simplu, dar nu totdeauna este eficient.

Pentru moment ne referim la prima soluţie, iar soluţia a doua va fi tratată într-un capitol nou cu privire la conversii.

Pentru a legaliza expresii de forma:

a+k

unde *a* este un obiect de tip *complex*, iar *k* o variabilă de tip *int*, putem adăuga o nouă funcţie *operator* membru sau prieten de felul celor de mai jos.

```
complex complex::operator + (int k)
/*returneaza obiectul de tip complex rezultat prin adaugarea la obiectul
  complex curent a valorii lui k */
{
 complex ztemp;

 ztem.real = real + k;
 ztemp.imag = imag;
 return ztemp;
}
```

În locul acestei funcţii membru putem defini funcţia prieten, de mai jos, care are acelaşi efect.

```
complex operator + (complex& z1,int k)
/* returneaza obiectul de tip complex rezultat prin adăugarea la z1 a
   valorii lui k */
```

```
{
 complex ztemp;

 ztemp.real = z1.real + k;
 ztemp.imag = z1.imag;
 return ztemp;
}
```

Adăugînd una din aceste funcţii, se pot utiliza instrucţiuni de forma:

```
complex a,b;
int k;
...
b = a+k;
b = a+123;
```

Să observăm că funcţiile de mai sus implică faptul că primul operand al operatorului + să fie de tip *complex*. Aceasta înseamnă că expresiile de forma:

```
k+a
123+a
```

sînt ilegale. Pentru a accepta astfel de expresii este necesar ca operatorul + să fie supraîncărcat în mod corespunzător, adică pentru expresii în care primul operand este de tip *int*, iar cel de al doilea de tip *complex*.

O astfel de supraîncărcare se poate realiza dar numai printr-o funcţie prieten deoarece în cazul funcţiilor membru, primul parametru este totdeauna obiectul curent şi deci acesta nu poate fi de tip *int*.

De aceea, pentru a admite expresii în care primul operand este de tip *int*, iar cel de al doilea este un obiect de tip *complex*, vom utiliza funcţia prieten de mai jos.

```
complex operator + (int k,complex& z)
/* returneaza obiectul complex k+z */
{
 complex ztemp;

 ztemp.real = z.real+k;
 ztemp.imag = z.imag;
 return ztemp;
}
```

Din acest exemplu se vede că supraîncărcarea operatorilor nu permite moştenirea proprietăţii de comutativitate a acestora. Astfel, deşi operatorul + este comutativ pentru operanzi de tipuri predefinite, supraîncărcarea lui

pentru expresii de forma:

a+k

unde:

a - Este obiect de tip *complex*.

k - Este o variabilă de tip *int*, nu este valabilă şi pentru expresii de forma:

k+a ,

Exerciţii:

23.1 Să se definească tipul abstract *complex* supraîncărcînd operatori pentru operaţii cu numere complexe după cum urmează:

operatori	**tipul operandului stîng**	**tipul operandului drept**	**tipul rezultatului**
Adunare			
+	complex	complex	complex
+	complex	double	complex
+	double	complex	complex
Scădere			
-	complex	complex	complex
-	complex	double	complex
-	double	complex	complex
Înmulţire			
*	complex	complex	complex
*	complex	double	complex
*	double	complex	complex
Împărţire			
/	complex	complex	complex
/	complex	double	complex

(continuare)

operatori	tipul operandului stîng	tipul operandului drept	tipul rezultatului
/	double	complex	complex
Ridicare la putere			
^	complex	int	complex
Test de egalitate			
==	complex	complex	int
==	complex	double	int
==	double	complex	int
Negativare			
-	complex		complex
Incrementare			
++	complex		complex
Decrementare			
--	complex		complex
Rădăcina pătrată			
!	complex		complex

De asemenea, se definesc funcţii membru pentru:

- modulul unui număr complex;
- argumentul unui număr complex;
- citirea unui număr complex;
- afişarea unui număr complex;
- incrementarea părţii imaginare a unui număr complex;
- decrementarea părţii imaginare a unui număr complex;

– returnarea părţii reale a unui număr complex;
– returnarea părţii imaginare a unui număr complex.

Ridicarea la putere (operatorul ^) se realizează folosind formula lui Moivre.

Dacă z = a+b*i, atunci fie forma trigonometrică a lui *z*:

z = r*(cos(alfa)+i*sin(alfa))

Pentru *n* întreg şi pozitiv, avem:

z**n = r**n*(cos(n*alfa)+i*sin(n*alfa))

Unghiul n*alfa se poate reduce la primul cerc dacă se pune sub forma:

n*alfa = 2*k*PI+beta

unde:

k - Este partea întreagă a împărţirii (n*alfa)/(2*PI).

PI = 3.14159265358979.

Dacă n < 0, atunci:

z**n = 1/z**abs(n)

Întrucît operatorul ** nu există în limbajul C, pentru ridicarea la putere s-a ales operatorul ^. Prioritatea acestuia este mai mică decît a operatorilor binari aritmetici. De aceea, în expresii mai complexe se vor utiliza paranteze rotunde pentru a impune ordinea dorită a operaţiilor.

De exemplu, ordinea operaţiilor în evaluarea expresiei:

a*x^10

unde *a* şi *x* sînt numere *complexe*, este următoarea:

- se înmulţeşte *a* cu *x*;
- rezultatul înmulţirii se ridică la puterea 10.

Pentru a realiza întîi ridicarea la putere a lui *x* şi apoi înmulţirea cu *a*, vom utiliza expresia:

a*(x^10)

Testul de egalitate este util mai ales la stabilirea dacă un număr *complex* este nul sau nu. O expresie de forma:

a == b

unde *a* şi *b* sînt ambele numere complexe sau unul este complex şi celălalt de tip numeric (*int*, *long*, *unsigned*, *double* etc.) are ca rezultat una din valorile 0

sau 1. Rezultatul expresiei este 0 dacă cei doi operanzi sînt diferiţi şi 1 altfel.

Operatorul de incrementare (++), incrementează partea reală a numărului complex la care se aplică.

În mod analog, operatorul de decrementare (--), decrementează partea reală a numărului complex la care se aplică.

Rădăcina pătrată se calculează folosind forma trigonometrică a numărului complex.

Astfel, dacă:

z = a+ib = r*(cos(alfa)+i*sin(alfa))

atunci rădăcina pătrată din z se determină ca mai jos:

z1 = sqrt(r)*(cos(alfa/2)+i*sin(alfa/2))

şi

z2 = sqrt(r)*(cos(alfa/2+PI)+i*sin(alfa/2+PI)) =

-sqrt(r)*(cos(alfa/2)+i*sin(alfa/2)) = -z1

FIŞIERUL BXXIII1.H

```
class complex {
    double real;
    double imag;

    static int citdouble(char *s,double& d);
    /* - afiseaza s;
       - citeste un numar si-l atribuie datei referite de d;
       - returneaza:
              0 - la sfirsit de fisier;
              1 - altfel. */
  public:                   // constructor
    complex(double x=0,double y=0);
    // modulul numarului complex

    double modul();
    // argumentul numarului complex

    double arg();
    // citeste un numar complex

    int citcomplex (char *s);
    /* - afiseaza s;
```

- citeste componentele numarului complex;
  - returneaza:
    0 - la sfirsit de fisier;
    1 - altfel. */

```
// afiseaza un numar complex

void afiscomplex(char *f);
/*afiseaza componentele numarului complex conform
  formatului f */

// incrementeaza partea imaginara
void ipi();

// decrementeaza partea imaginara
void dpi();

// obiectul curent se noteaza cu z
// obiectul rezultat se noteaza cu r
// operatori unari

// negativare
complex operator - ();  // r = -z

// incrementeaza partea reala
complex operator ++ ();

// decrementeaza partea reala
complex operator -- ();

// radacina patrata
complex operator !();  // r = sqrt(z)

// operatori binari
// adunare

// r = z+z2
complex operator + (complex& z2);

// r = z+d
complex operator + (double d);
```

```
//r = d+z2
friend complex operator + (double d,
                           complex& z2);

//scadere
//r = z-z2
complex operator - (complex& z2);

//r = z-d
complex operator - (double d);

//r = d-z2
friend complex operator - (double d,
                           complex& z2);

//inmultire
//r = z*z2
complex operator * (complex& z2);

//r = z*d
complex operator * (double d);

//r = d*z2
friend complex operator * (double d,
                           complex& z2);

//impartire
//r = z/z2
complex operator / (complex& z2);

//r = z/d
complex operator / (double d);

//r = d/z2
friend complex operator / (double d,
                           complex& z2);

//ridicare la putere
//r = z**i
complex operator ^ (int i);
```

```
// test de egalitate

/* a == b; se returneaza 0 sau 1 dupa cum relatia
   este falsa sau adevarata */

//z == b
int operator == (complex& b);

//z == d
int operator == (double d);

//d == b
friend int operator == (double d,complex& b);

// returneaza partea reala a lui z
double retreal();

// returneaza partea imaginara a lui z
double retimag();
};
```

Funcţiile care intervin în definiţia clasei *complex* se definesc într-un fişier separat, de extensie *CPP*.

FIŞIERUL BXXIII1

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __MATH_H
#include <math.h>
#define __MATH_H
#endif
#ifndef __PI
#define PI 3.14159265358979
#define __PI
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#ifndef __BXXIII1_H
#include "BXXIII1.H"
#define __BXXIII1_H
```

```
#endif
#ifndef __STRING_H
#include <string.h>
#define __STRING_H
#endif

inline complex::complex(double x,double y)
// constructor utilizat la initializare;
// implicit x=y=0
{
 real = x;
 imag = y;
}

inline double complex::modul()
// returneaza modulul obiectului curent
{
 return sqrt(real*real + imag*imag);
}

double complex::arg()
// returneaza argumentul obiectului curent
{
 if(real == 0 && imag == 0)
     return 0.0;  // z == 0
 if(imag == 0)     // z = parte reala
     if(real > 0)
         return 0.0;   // z = parte reala > 0
     else
         return PI;    // z = parte reala < 0
 if(real == 0)      // z = parte imaginara
     if(imag > 0)
         return PI/2;         // z = parte imaginara > 0
     else
         return (3*PI)/2;   // z = parte imaginara < 0

 // z = real +i*imag; real != 0 si imag != 0

 double a = atan(imag/real);
 if(real < 0)
     return PI+a;    // real < 0 si imag != 0
 if(imag < 0)
```

```
    return 2*PI+a;  // real > 0 si imag < 0
 return a;           // real > 0 si imag > 0
}

int complex::citdouble(char *s,double& d)
/* - afiseaza s;
   - citeste un numar si-l atribuie datei referite de d;
   - returneaza:
        0 - la sfirsit de fisier;
        1 - altfel.
*/
{
 char t[255];

 for(;;){
     if(strlen(s))   // sirul s nu este vid
        printf(s);
     if(gets(t) == 0)
        return 0;    // s-a intilnit EOF
     if(sscanf(t,"%lf",&d) == 1)
        break;
     printf("nu s-a tastat un numar\n");
     printf("se reia citirea\n");
 }
 return 1;
}

int complex::citcomplex(char *s)
/* - afiseaza s daca nu este vid; daca s este vid se afiseaza texte standard:
       Partea reala si Partea imaginara
   - citeste componentele obiectului complex curent;
   - returneaza:
       0 - la sfirsit de fisier;
       1 - altfel.
*/
{
 if(*s){   // sirul s nu este vid;

   // se citesc componentele obiectului curent
     for(;;){
        printf(s);
        int i;
        double d;
```

```
        if((i = scanf("%lf",&d)) == EOF)
            return 0;
        if(i != 1){
            printf("nu s-a tastat un numar\n");
            printf("pentru partea reala\n");
            printf("se reia citirea de la\
                    inceput\n");
            fflush(stdin);  //videaza zona tampon
            continue;
        }
        else
            real = d;
        if((i = scanf("%lf",&d)) == EOF)
            return 0;
        if(i != 1){
            printf("nu s-a tastat un numar\n");
            printf("pentru partea imaginara\n");
            printf("se reia citirea de la\
                    inceput\n");
            fflush(stdin);  //videaza zona tampon
        }
        else{
            imag = d;
            fflush(stdin);
            return 1;
        }
    } //sfirsit for
  } //sfirsit if(s)

// s pointeaza spre sirul vid

  if(complex::citdouble("Partea reala = ",real) ==
      0)
    return 0;
  if(complex::citdouble("Partea imaginara = ",
      imag) == 0)
    return 0;
  return 1;
}
```

```
void complex::afiscomplex(char *f)
/* - afiseaza componentele obiectului curent cu formatul f;
   - daca f reprezinta sirul vid, componentele se afiseaza in format
     standard */
{
 if(*f)   // formatul de afisare nu este vid
     printf(f,real,imag);
 else     // format standard
     printf("%g+i*(%g)\n",real,imag);
}

inline void complex::ipi()
/* incrementeaza partea imaginara a obiectului complex curent */
{
 imag++;
}

inline void complex::dpi()
/* decrementeaza partea imaginara a obiectului complex curent */
{
 imag--;
}

complex complex::operator - ()
// returneaza negativul obiectului complex curent: r = -z
{
 complex r;

 r.real = -real;
 r.imag = -imag;
 return r;
}

inline complex complex::operator ++ ()
/* incrementeaza partea reala a obiectului complex curent */
{
 real++;
 return *this;
}
```

```
inline complex complex::operator -- ()
/* decrementeaza partea reala a obiectului complex curent */
{
 real--;
 return *this;
}

complex complex::operator ! ()
/* calculeaza radacina patrata din obiectul complex curent */
{
 complex r;   // implicit real=imag=0
 double d;

 if((d=modul()) == 0)
     return r;   // obiect complex nul

 double alfa;
 alfa = arg();   // argumentul obiectului complex
 d = sqrt(d);    // radacina patrata din modul
 alfa = alfa/2;
 r.real = d*cos(alfa);
 r.imag = d*sin(alfa);
 return r;
}

complex complex::operator + (complex& z2) // r = z+z2
{
 complex r;

 r.real = real + z2.real;
 r.imag = imag + z2.imag;
 return r;
}

complex complex::operator + (double d)    // r = z+d
{
 complex r;

 r.real = real + d;
 r.imag = imag;
 return r;
}
```

```
complex operator + (double d,complex& z2) //r = d+z2
{
 complex r;

 r.real = d + z2.real;
 r.imag = z2.imag;
 return r;
}

complex complex::operator - (complex& z2)  //r = z-z2
{
 complex r;

 r.real = real - z2.real;
 r.imag = imag - z2.imag;
 return r;
}

complex complex::operator - (double d)     //r = z-d
{
 complex r;

 r.real = real-d;
 r.imag = imag;
 return r;
}

complex operator - (double d,complex& z2)  //r = d-z2
{
 complex r;

 r.real = d - z2.real;
 r.imag = -z2.imag;
 return r;
}

complex complex::operator * (complex& z2)   //r = z*z2
{
 complex r;

 r.real = real*z2.real - imag*z2.imag;
 r.imag = real*z2.imag + imag*z2.real;
 return r;
```

```
}

complex complex::operator * (double d)  //r = z*d
{
 complex r;

 r.real = real*d;
 r.imag = imag*d;
 return r;
}

complex operator * (double d,complex& z2) //r = d*z2
{
 complex r;

 r.real = d * z2.real;
 r.imag = d * z2.imag;
 return r;
}

complex complex::operator / (complex& z2)
// r = z/z2; r = 0 daca z2 = 0
{
 complex r;
 double d = z2.real*z2.real + z2.imag*z2.imag;

 if(d == 0)  //z2 = 0
     return r;
 r.real = (real*z2.real+imag*z2.imag)/d;
 r.imag = (imag*z2.real - real*z2.imag)/d;
 return r;
}

complex complex::operator / (double d)
// r = z/d; r = 0 daca d = 0
ă
 complex r;

 if(d == 0)
    return r;
 r.real = real/d;
 r.imag = imag/d;
 return r;
```

```
}

complex operator / (double d,complex& z2)
// r = d/z2; r = 0 daca z2 = 0
{
 complex r;
 double dm = z2.real*z2.real + z2.imag*z2.imag;

 if(dm == 0)  // z2 = 0
    return r;
 r.real = z2.real*d/dm;
 r.imag = -z2.imag*d/dm;
 return r;
}

complex complex::operator ^ (int i)
// r = z**i; r = 0 daca z = 0
{
 complex r;  // real = imag = 0
 double d = modul();   // modulul obiectului curent
 if(d == 0)
    return r;   // obiectul curent este nul; ridicat la
                // puterea i rezulta obiectul nul

 int n =i < 0 ? -i : i;  // n = abs(i)

 if(n == 0)  // z**0 = 1
    return complex(1,0);
 if(i == 1)  // z**1 = z
    return *this;
 if(i == -1)  // z**(-1) = 1/z
    r = *this;
 if(n > 1){
    d = pow(d,(double)n);   // d = d**n
    double a = arg();   // a = argumentul obiectului
                        // complex curent
    a *= n;    // a argument*n

    // reducere la primul cerc

    double doipi=2*PI;
    long x = a/doipi;
```

```
    a -= x*doipi;
    r.real = d*cos(a);
    r.imag = d*sin(a);
 }
 if(i < 0)  // x**i = 1/z**n
    r = 1.0/r;   //operatorul / este supraincarcat
                 // pentru double/complex
 return r;
}

inline int complex::operator == (complex& z2)
/* returneaza:
     1 daca z=z2;
     0 altfel.
*/
{
 return real == z2.real && imag == z2.imag;
}

inline int complex::operator == (double d)
/* returneaza:
     1 daca z=d;
     0 altfel.
*/
{
 return real == d && imag == 0;
}

inline int operator == (double d,complex& z2)
/* returneaza:
     1 daca d=z2;
     0 altfel.
*/
{
 return d == z2.real && z2.imag == 0;
}

inline double complex::retreal()
/* returneaza partea reala a obiectului complex curent */
{
 return real;
}
```

```
inline double complex::retimag()
/* returneaza partea imaginara a obiectului complex curent */
{
 return imag;
}
```

23.2 Să se scrie un program care rezolvă ecuaţii de gradul întîi cu coeficienţii complecşi:

ax+b = 0 (a şi b sînt numere complexe)

Programul citeşte numerele complexe *a* şi *b* şi afişează valoarea lui *x* dacă a != 0. Se pot rezolva mai multe ecuaţii de această formă. Pentru a întrerupe execuţia programului se tastează sfîrşitul de fişier.

PROGRAMUL BXXIII2

```
#include "BXXIII1.CPP"

main()
/* - rezolva ecuatii de gradul 1 cu coeficienti complecsi de forma:
      ax + b = 0;
   - programul se termina la tastarea sfirsitului de fisier.
*/
{
 complex a,b,x;

 for(;;){
     if(a.citcomplex("a= ") == 0)
        exit(0);
     if(b.citcomplex("b= ") == 0)
        exit(1);
     if(a == 0.0 && b == 0.0){
        printf("ecuatie nedeterminata\n");
        continue;
     }
     if(a == 0.0){
        printf("ecuatia nu are solutie\n");
        continue;
     }
     x = -b/a;
     x.afiscomplex("x = %g+i*(%g)\n");
 } //sfirsit for
}
```

Observaţii:

1. Instrucţiunea:

   x = -b/a;

   conţine doi operatori supraîncărcaţi:

   – operatorul unar -

   şi

   – operatorul binar /

   Operatorul unar este prioritar celui binar.

2. La lansarea programului se afişează:

   a=

   apoi, se aşteaptă tastarea componentelor numărului complex *a*.

   Dacă nu se tastează EOF, după tastarea componentelor numărului *a*, se va afişa:

   b=

   apoi, se aşteaptă tastarea componentelor numărului complex *b*.

23.3 Să se scrie un program care citeşte numere complexe şi le afişează cu partea reală mărită cu 1.

PROGRAMUL BXXIII3

```
#include "BXXIII1.CPP"

main()
/* - citeste numere complexe si le afiseaza cu partea reala marita cu 1;
   - programul se termina la intilnirea sfirsitului de fisier.
*/
{
 complex a;

 for(;;){
    if(a.citcomplex("") == 0)
       exit(0);
    a++;    // incrementeaza partea reala a lui a
    a.afiscomplex("");
 } // sfirsit for
}
```

Observaţii:

1. Funcţia *citcomplex* este apelată cu şirul vid. În acest caz se afişează textul:

   Partea reala =

   şi după tastarea numărului respectiv se afişează textul:

   Partea imaginara =

2. Partea reală a numărului complex se incrementează apelîndu-se operatorul unar ++ supraîncărcat pentru numere complexe. În program s-a folosit forma postfixată. Se poate utiliza la fel de bine forma prefixată:

   ++a;

   obţinîndu-se acelaşi efect.

Menţionăm că nu se face nici o diferenţă între formele postfixate şi prefixate ale operatorilor de incrementare şi decrementare suparaîncărcaţi pentru tipuri abstracte.

23.4 Să se scrie un program care rezolvă ecuaţii binome cu coeficienţi complecşi.

Prin ecuaţie binomă înţelegem o ecuaţie de forma:

a*x**n+b = 0

unde:

n - Este un număr natural pozitiv.

a şi b - Sînt numere complexe.

Pentru n > 1, soluţia ecuaţiei este rădăcina de ordinul *n* din numărul complex -b/a.

Rădăcina de ordinul *n* dintr-un număr complex se determină din forma trigonometrică a acestuia.

Fie:

z=x+i*y un număr complex

şi

z=r*(cos(alfa)+i*sin(alfa)) forma lui trigonometrică.

Atunci rădăcinile de ordinul *n* din *z* se calculează ca mai jos:

zk = r**(1/n)*(cos((alfa+2*k*PI)/n)+
     i*sin((alfa+2*k*PI)/n)), pentru k=0,1,2,...,n-1.

Expresia r**(1/n) are ca valoare rădăcina de ordinul *n* din numărul raţional r. Valoarea ei se determină cu ajutorul funcţiei *pow*.

PROGRAMUL BXXIII4

```
#include "BXXIII1.CPP"

main()
/* - rezolva ecuatii binome(a*x**n + b = 0);
   - programul se termina la tastarea sfirsitului de fisier. */
{
 complex a,b,x;
 int n;

 for(;;){
    if(a.citcomplex("a= ") == 0)
       exit(0);
    if(b.citcomplex("b= ") == 0)
       exit(1);
    if(a == 0.0 && b == 0.0){
       printf("ecuatie nedeterminata\n");
       continue;
    }
    if(a == 0.0){
       printf("ecuatia nu are solutie\n");
       exit(1);
    }

    //a != 0

    int i;

    for(;;){
       printf("gradul ecuatiei = ");
       if((i=scanf("%d",&n)) == EOF){
          printf("s-a tastat EOF\n");
          exit(1);
       }
       if(i == 1 && n > 0)
          break;
       printf("nu s-a tastat un intreg\
             pozitiv\n");
       fflush(stdin);
    }
```

```
        //n > 0

        x = -b/a;

        if(n > 1 && !(x == 0)) {

            //ecuatie de grad mai mare ca 1

            double r = x.modul();
            double teta = x.arg();
            r = pow(r,1.0/n);
            for(int k = 0; k < n; k++){
                double alfa = (teta + 2*PI*k)/n;
                complex xk(r*cos(alfa),r*sin(alfa));
                printf("x%d=",k);
                xk.afiscomplex("");
            }
        }
        else      //ecuatie de gradul intii sau solutie nula
            x.afiscomplex("x=%g+i(%g)\n");
    } //sfirsit for
}
```

23.5 Să se scrie un program care rezolvă ecuaţii de gradul 2 cu coeficienţi complecşi.

PROGRAMUL BXXIII5

```
#include "BXXIII1.CPP"

main()
/* rezolva ecuatii de gradul 2 cu coeficientii complecsi:
      a*x*x + b*x + c = 0
*/
{
 complex a,b,c,x;

 for(;;){
    if(a.citcomplex("a= ") == 0)
       exit(0);
    if(b.citcomplex("b= ") == 0)
       exit(1);
    if(c.citcomplex("c= ") == 0)
       exit(1);
```

```
if(a == 0.0 && b == 0.0 && c == 0.0){
    printf("ecuatie nedeterminata\n");
    continue;
}
if(a == 0.0 &&b == 0.0){
    printf("ecuatia nu are solutie\n");
    continue;
}
if(a == 0.0){
    printf("ecuatie de gradul 1\n");
    x = -c/b;
    x.afiscomplex("x = %g+i*(%g)\n");
    continue;
}

// a != 0: ecuatie de gradul 2

complex doia = 2.0*a;
complex delta = b*b-4.0*a*c;
delta =! delta;   // radacina patrata din delta
x = (-b+delta)/doia;
x.afiscomplex("x1 = %g+i*(%g)\n");
x = (-b-delta)/doia;
x.afiscomplex("x2 = %g+i*(%g)\n");
} // sfirsit for
}
```

23.6 Să se scrie un program care citeşte numere complexe şi afişează puterea a 5-a a fiecărui număr citit, adunat cu 1.

Programul foloseşte operatorul ^ care a fost supraîncărcat cu funcţia putere pentru numere complexe. Dacă *z* este numărul complex citit, atunci programul afişează numărul complex definit de expresia:

z*z*z*z*z+1

care, cu ajutorul operatorului ^ se scrie:

(z^5)+1

PROGRAMUL BXXIII6

```
#include "BXXIII1.CPP"
```

```
main()
/* - citeste numere complexe;
   - afiseaza puterea a 5-a a fiecarui numar citit adunat cu 1
*/
{
 complex z,v;

 for(;;){
     if(z.citcomplex("") == 0)
        exit(0);
     v =(z^5) +1;
     z.afiscomplex("z = %g+i*(%g)");
     v.afiscomplex(" z**5+1 = %g+i*(%g)\n");
 }
}
```

Observaţie:

În absenţa parantezelor, expresia:

z^5+1

ridică pe *z* la puterea a şasea, deoarece operatorul + este mai prioritar decît ^.

23.7 Numim *vector* n-dimensional un sistem ordonat de *n* numere reale. Se cere să se implementeze conceptul de *vector bidimensional*. În acest scop, vom presupune că vectorul bidimensional este un sistem ordonat de 2 numere de tip *double*.

Asupra vectorilor vom defini următoarele operaţii:

- suma a doi vectori;
- diferenţa a doi vectori;
- produsul dintre un vector şi un număr;
- negativarea unui vector;
- lungimea unui vector;
- produsul scalar a doi vectori;
- citirea şi afişarea componentelor vectorului;
- returnarea componentelor unui vector.

FIŞIERUL BXXIII7.H

```
class vb {
     double c1,c2;

  public:
     // constructor
     vb(double v1=0, double v2=0);

     // suma a doi vectori: r=vb+vb2
     vb operator + (vb& vb2);

     // diferenta a doi vectori: r=vb-vb2
     vb operator - (vb& vb2);

     // produsul dintre un vector si un numar: r=vb*a
     vb operator * (double a);

     // produsul dintre un numar si un vector: r=a*vb2
     friend vb operator * (double a, vb& v2);

     // negativarea unui vector: r = -vb
     vb operator -();

     // lungimea vectorului curent
     double lungime();

     // produsul scalar a 2 vectori r=(vb,vb2)
     double operator *(vb& vb2);

     /* - citeste componentele unui vector, dupa afisarea unui text;
        - returneaza:
                0 - la sfirsit de fisier;
                1 - altfel.
     */
     int citvb(char *text);

     /* afiseaza componentele unui vector cu un format
           standard: (c1,c2) */
     void afisvb();

     /* afiseaza componentele unui vector cu un format dat */
     void afisvb(char *format);
```

```
    // returneaza prima componenta a vectorului
    double retc1();

    // returneaza a doua componenta a vectorului
    double retc2();
};
```

Funcţiile membru şi operatorii se definesc în fişierul de mai jos, de extensie *cpp*.

FIŞIERUL BXXIII7

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __MATH_H
#include <math.h>
#define __MATH_H
#endif
#ifndef __STRING_H
#include <string.h>
#define __STRING_H
#endif
#ifndef __BXXIII7_H
#include "BXXIII7.H"
#define __BXXIII7_H
#endif

inline vb::vb(double x,double y)     // constructor
{
 c1 = x;
 c2 = y;
}

vb vb::operator + (vb& vb2) // suma a doi vectori r = vb+vb2
{
 vb vtemp;

 vtemp.c1 = c1 + vb2.c1;
 vtemp.c2 = c2 + vb2.c2;
 return vtemp;
}
```

```
vb vb::operator - (vb& vb2)// diferenta a doi vectori r=vb-vb2
{
 vb vtemp;

 vtemp.c1 = c1 - vb2.c1;
 vtemp.c2 = c2 - vb2.c2;
 return vtemp;
}

vb vb::operator * (double a)
// produsul dintre un vector si un numar
{
 vb vtemp;

 vtemp.c1 = a*c1;
 vtemp.c2 = a*c2;
 return vtemp;
}

inline vb operator *(double a,vb& vb2)
// produsul dintre un numar si un vector
{
 return vb2*a;

 /* operatorul * este supraincarcat pentru produsul dintre un vector si
    un numar prin functia precedenta */

}

vb vb::operator -()        // negativarea unui vector r = -vb
{
 vb vtemp;

 vtemp.c1 = -c1;
 vtemp.c2 = -c2;
 return vtemp;
}

inline double vb::lungime()   // lungimea vectorului curent
{
 return sqrt(c1*c1+c2*c2);
}
```

```
inline double vb::operator*(vb& vb2)
// produsul scalar a doi vectori
{
 return c1*vb2.c1+c2*vb2.c2;
}

int vb::citvb(char *text)
/* - citeste componentele unui vector dupa afisarea unui text
     definit de parametru;
   - returneaza:
          0 - la intilnirea sfirsitului de fisier;
          1 - altfel.
*/
{
 char t[255];
 double d;

 for(;;) {
     if(strlen(text))
        printf(text);
     printf("\n prima componenta a vectorului = ");
     if(gets(t) == 0)
        return 0;    //s-a intilnit EOF
     if(sscanf(t,"%lf",&d) != 1) {
        printf("nu s-a tastat un numar\n");
        printf("se reia citirea\n");
        continue;
     }
     c1 = d;
     printf("componenta a doua a vectorului = ");
     if(gets(t) == 0)
        return 0;  //s-a intilnit EOF
     if(sscanf(t,"%lf",&d) == 1){
        c2 = d;
        return 1;
     }
     printf("nu s-a tastat un numar\n");
     printf("se reia citirea\n");
 }
}
```

```
inline void vb::afisvb()
/* afiseaza componentele vectorului curent folosind un
   format standard */
{
 printf("vb = (%g,%g)\n",c1,c2);
}

void vb::afisvb(char *format)
/* afiseaza componentele vectorului curent folosind formatul
   definit de parametru */
{
 if(format && strlen(format))
     printf(format,c1,c2);
 else
    /* afisarea cu format standard deoarece parametrul
       format este fie nul, fie pointeaza spre sirul vid */

     afisvb();   // afiseaza cu format standard
}

inline double vb::retc1()
// returneaza prima componenta a vectorului
{
 return c1;
}

inline double vb::retc2()
// returneaza componenta a doua a vectorului
{
 return c2;
}
```

23.8 Să se scrie un program care citeşte 2 perechi de numere ce reprezintă componentele vectorilor *a* şi *b*, apoi calculează şi afişează:

- lungimea fiecărui vector;
- suma: a+b;
- diferenta: a-b;
- suma: 3*a+7*b;
- suma: -a-b;
- produsul scalar al celor doi vectori: (a,b).

PROGRAMUL BXXIII8

```
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#include "BXXIII7.CPP"

main()
/* - citeste componentele vectorilor a si b;
   - calculeaza si afiseaza:
   - lungimea vectorilor a si b;
   - suma: a+b;
   - diferenta: a-b;
   - suma: 3*a+7*b;
   - suma: -a-b;
   - produsul scalar: (a,b).
*/
{
 vb a,b;
 char er[] = "s-a intilnit EOF\n";

 if(a.citvb("vectorul a\n") == 0){
     printf(er);
     exit(1);
 }
 if(b.citvb("vectorul b\n") == 0){
     printf(er);
     exit(1);
 }

 // afiseaza lungimile vectorilor
 printf("lungime(a) = %g\n", a.lungime());
 printf("lungime(b) = %g\n", b.lungime());

 // afiseaza valorile celorlalte expresii
 vb c;
 c=a+b;
 c.afisvb("a+b = %g\t %g\n");
 c=a-b;
 c.afisvb("a-b = %g\t %g\n");
 c=3*a+7*b;
 c.afisvb("3*a+7*b = %g\t %g\n");
 c=-a-b;
```

```
   c.afisvb("-a-b = %g\t %g\n");

   // afiseaza produsul scalar
   printf("(a,b) = %g\n",a*b);
}
```

23.9 Să se definească tipul abstract care implementează conceptul de matrice de ordinul 2.

La definirea unei matrice pătratice de ordinul doi se poate proceda în mai multe moduri:

- tablou bidimensional de ordinul 2*2;
- tablou unidimensional de două elemente, fiecare element fiind un obiect de tip vector bidimensional;
- două obiecte de tip vector bidimensional.

În exerciţiul de faţă, matricea se implementează printr-un sistem ordonat de două obiecte de tip vector bidimensional. Primul obiect defineşte prima linie a matricei, iar cel de al doilea obiect defineşte linia a doua a matricei.

Asupra matricelor pătratice de ordinul doi se definesc următoarele operaţii:

- suma a două matrice;
- diferenţa a două matrice;
- produsul a două matrice;
- inversa unei matrice;
- negativarea unei matrice;
- produsul unei matrice cu un număr;
- calculul determinantului unei matrice;
- produsul dintre o matrice şi un vector bidimensional.

Alte funcţii membru:

- citirea elementelor unei matrice;
- afişarea elementelor unei matrice;
- returnarea liniilor unei matrice.

FIŞIERUL BXXIII9.H

```
#ifndef __BXXIII7_H
#include "BXXIII7.H" //clasa vb defineste vectorii bidimensionali
#define __BXXIII7.H
#endif

class mpdoi {
     vb linia1;
     vb linia2;
  public:
     //constructor

     mpdoi(double c11=0,double c12=0,double c21=0,
           double c22=0);

     //functii membru

     /* - citeste elementele unei matrice;
        - returneaza:
              0 -la intilnirea sfirsitului de fisier;
              1 - altfel.
     */
     int citmpdoi(char *text);

      /* afiseaza elementele matricei curente folosind un format
         standard */
     void afismpdoi();

     /* afiseaza elementele matricei curente folosind formatul format1
       pentru linia unu si format2 pentru linia doi */
     void afismpdoi(char *format1, char *format2);

     //returneaza linia intiia a matricei
     vb retlin1();

     //returneaza linia a doua a matricei
     vb retlin2();

     //operatori

     //suma a doua matrice
     mpdoi operator + (mpdoi& mpdoi2);
```

```
    // diferenta a doua matrice
    mpdoi operator - (mpdoi& mpdoi2);

    // produsul a doua matrice
    mpdoi operator * (mpdoi& mpdoi2);

    // inversa unei matrice
    mpdoi operator !();

    // negativarea unei matrice
    mpdoi operator -();

    // produsul unei matrice cu un numar
    mpdoi operator * (double a);

    // produsul unei matrice cu un vector
    vb operator *(vb& vb2);

    // calculul determinantului unei matrice
    double operator ~();
};
```

Funcţiile membru şi supraîncărcarea operatorilor se definesc într-un fişier separat, de extensie *cpp*.

Amintim că inversa matricei de ordinul 2:

a11 a12
a21 a22

este matricea:

a22/d -a12/d
-a21/d a11/d

unde prin *d* s-a notat determinantul matricei iniţiale.

În comentariile funcţiilor definite mai jos vom folosi următoarele notaţii:

r - matricea rezultat;
mpdoi - matricea curentă.

FIŞIERUL BXXIII9

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __BXXIII7_CPP
#include "BXXIII7.CPP"
#define __BXXIII7_CPP
#endif
#ifndef __BXXIII9_H
#include "BXXIII9.H"
#define __BXXIII9_H
#endif

// constructor

inline mpdoi::mpdoi(double c11,double c12,
        double c21,double c22):linia1(c11,c12),
        linia2(c21,c22)
{
}

// functii membru

int mpdoi::citmpdoi(char *text)
/* - afiseaza textul spre care pointeaza parametrul text;
   - citeste elementele matricei curente;
   - returneaza:
        0 - la intilnirea sfirsitului de fisier;
        1 - altfel. */
{
 if(text && strlen(text))
    printf(text);
 if(linia1.citvb("\n prima linie\n") == 0 )
    return 0;
 if(linia2.citvb("\n linia a doua\n") == 0 )
    return 0;
}

void mpdoi::afismpdoi()
/* afiseaza matricea curenta folosind un format standard */
{
 linia1.afisvb();
```

```
 linia2.afisvb();
}

void mpdoi::afismpdoi(char *format1,char *format2)
/* afiseaza matricea curenta folosind formatele:
   format1 - pentru prima linie;
   format2 - pentru linia a doua
*/
{
 linia1.afisvb(format1);
 linia2.afisvb(format2);
}

inline vb mpdoi::retlin1()
// returneaza prima linie a matricei curente
{
 return linia1;
}

inline vb mpdoi::retlin2()
// returneaza linia a doua a matricei curente
{
 return linia2;
}

// operatori

mpdoi mpdoi::operator + (mpdoi& mpdoi2)
// r = mpdoi+mpdoi2
{
 mpdoi rtemp;

 rtemp.linia1 = linia1+mpdoi2.linia1;
 rtemp.linia2 = linia2+mpdoi2.linia2;
 return rtemp;
}

mpdoi mpdoi::operator - (mpdoi& mpdoi2)
// r = mpdoi-mpdoi2
{
 mpdoi rtemp;

 rtemp.linia1 = linia1-mpdoi2.linia1;
```

```
 rtemp.linia2 = linia2-mpdoi2.linia2;
 return rtemp;
}

mpdoi mpdoi::operator * (double a) // r = mpdoi*a
{
 mpdoi rtemp;

 rtemp.linia1 = linia1*a;
 rtemp.linia2 = linia2*a;
 return rtemp;
}

vb mpdoi::operator * (vb& vb2)  // r = mpdoi*vb2
{
 vb rtemp(linia1*vb2,linia2*vb2);
 return rtemp;
}

mpdoi mpdoi::operator * (mpdoi& mpdoi2)
// r = mpdoi*mpdoi2
{
 // coloana intii a matricei mpdoi2
 vb col1(mpdoi2.linia1.retc1(),
        mpdoi2.linia2.retc1());

 // coloana a doua a matricei mpdoi2
 vb col2(mpdoi2.linia1.retc2(),
        mpdoi2.linia2.retc2());

 // (r11,r21) = mpdoi * col1
 vb rvb1;
 rvb1 = *this*col1;

 // (r12,r22)
 vb rvb2;
 rvb2 = *this*col2;

 // matricea produs
 mpdoi rtemp(rvb1.retc1(),rvb2.retc1(),
             rvb1.retc2(),rvb2.retc2());
 return rtemp;
}
```

```
inline double mpdoi::operator~()
// calculeaza determinantul matricei curente
{
 return linia1.retc1()*linia2.retc2() -
       linia1.retc2()*linia2.retc1();
}

mpdoi mpdoi::operator -()  // r = -mpdoi
{
 mpdoi rtemp;

 rtemp.linia1 = -linia1;
 rtemp.linia2 = -linia2;
 return rtemp;
}

mpdoi mpdoi::operator !()
// returneaza inversa matricei curente
{
 double d = ~*this;   // calculeaza determinantul matricei
                      // curente
 if(d != 0){
    mpdoi rtemp(linia2.retc2()/d,
              -linia1.retc2()/d,
               -linia2.retc1()/d,
                linia1.retc1()/d);
    return rtemp;
 }
 printf("matricea are determinantul nul\n");
 mpdoi rtemp;
 return rtemp;   // matricea nula
}
```

23.10 Să se scrie un program care rezolvă un sistem de două ecuaţii liniare cu două necunoscute.

Fie sistemul:

(1) ax = b

unde:

a - Este o matrice pătratică de ordinul 2.

b - Este o matrice coloană de ordinul 2*1.

Matricea *b* corespunde conceptului de vector bidimensional, implementat cu ajutorul clasei *vb*, definită în exercițiul 23.7.

Soluția sistemului (1) există și este unică dacă determinantul matricei *a* este diferit de zero. Dacă notăm cu *inva* inversa matricei *a*, atunci soluția sistemului (1) se poate exprima astfel:

(2) x = inva*b

Programul din acest exercițiu citește elementele matricelor *a* și *b*, calculează soluția *x* cu ajutorul relației (2) și afișează soluția respectivă. Apoi,se determină vectorul reziduu *r* cu ajutorul relației:

(3) r = ax-b

unde:

x - Este soluția determinată mai sus.

În final, se afișează vectorul reziduu.

PROGRAMUL BXXIII10

```
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#include "BXXIII9.CPP"

main()
/* - citeste elementele matricelor a si b;
   - calculeaza si afiseaza solutia sistemului:
        ax = b
   - calculeaza si afiseaza vectorul reziduu:
        r = ax - b
*/
{
 mpdoi a;
 vb b,x,r;
 char er[] = "S-a tastat EOF\n";

 // citeste matricea a
 if(a.citmpdoi("matricea a\n") == 0){
     printf(er);
     exit(1);
 }
 if(~a == 0){    // determinantul nul
     printf("determinantul sistemului este nul\n");
```

```
    exit(1);
  }.

  // citeste vectorul b
  if(b.citvb("termenul liber\n") == 0){
     printf(er);
     exit(1);
  }

  // calculeaza solutia sistemului
  x = !a *b;    // x = inva*b

  // afiseaza solutia sistemului
  x.afisvb("x1 = %.15g\t x2 = %.15g\n");

  // calculeaza vectorul reziduu
  r = a*x-b;

  // afiseaza vectorul reziduu
  printf("vectorul reziduu\n");
  r.afisvb("linia1 = %.16g\t linia2 = %.16g\n");
}
```

23.1. Supraîncărcarea operatorilor new şi delete

În capitolul 20 s-a arătat că operatorii *new* şi *delete* pot fi utilizaţi pentru alocarea dinamică a datelor în memoria *heap*. Operatorul *new* alocă, la execuţie, o dată în memoria *heap*. Operatorul *delete* dezalocă (distruge) o dată care a fost alocată, în prealabil, prin operatorul *new*. Aceşti operatori pot fi utilizaţi atît pentru a gestiona date de tipuri predefinite cît şi date de tip abstract, adică obiecte.

Operatorul *new* are ca operand numele tipului datei pentru care se alocă memorie:

new *nume_tip*

Printr-o expresie de această formă se alocă în memoria *heap* o zonă de memorie necesară pentru a păstra o dată de tip *nume_tip*. Rezultatul aplicării operatorului *new* este adresa de început a zonei de memorie alocate. Această adresă este un pointer spre tipul *nume_tip*.

Exemple:

1. `int *p = new int;`

Se alocă o zonă de memorie necesară pentru a păstra o dată de tip *int*. Adresa de început a acestei zone reprezintă un pointer spre *int* şi se atribuie lui *p*.

2. `double *q = new double;`

Se alocă o zonă de memorie necesară pentru a păstra o dată de tip *double*.

Adresa de început a acestei zone reprezintă un pointer spre *double* şi se atribuie lui *q*.

3.

```
class vb {
        double c1,c2;
    public:
        vb()
        {
            c1 = 0;
            c2 = 0;
        }
        ...
};
vb *pvb = new vb;
```

Se alocă zonă de memorie pentru a păstra un obiect de tip *vb*. Adresa de început a acestei zone reprezintă un pointer spre *vb* şi se atribuie lui *pvb*.

La alocare, se apelează automat constructorul implicit.

Operatorul *new* se poate aplica pentru tipurile abstracte el avînd o supraîncărcare globală. Această supraîncărcare globală este numită *standard* sau *predefinită*.

În mod analog, operatorul *delete* are o supraîncărcare globală şi el poate fi utilizat pentru a dezaloca (distruge) datele pentru care s-a alocat memorie cu ajutorul lui *new*. Astfel, pentru a distruge datele alocate în exemplele de mai sus, nu avem decît să utilizăm expresiile:

- delete p; pentru data alocată în exemplul 1;
- delete q; pentru data alocată în exemplul 2;
- delete pvb; pentru obiectul alocat în exemplul 3.

Amintim că la distrugerea obiectelor se apelează destructorul clasei (cel definit de utilizator sau cel implicit dacă nu există un astfel de destructor).

Operatorul *new* se poate utiliza nu numai pentru a aloca memorie pentru o dată, ci şi pentru a iniţializa data respectivă.

Exemple:

1. `int *p = new int(10);`

   În zona alocată în memoria *heap* se păstrează valoarea 10 de tip *int*, iar *p* pointează spre zona respectivă.

2.

```
class vb {
        double c1,c2;
    public:
        vb()
        {
            c1 = 0;
            c2 = 0;
        }
        vb(double v1,double v2)
        {
            c1 = v1;
            c2 = v2;
        }
        ...
};
vb *pvb = new vb(1,2);
```

   În zona alocată în memoria *heap* se păstrează un obiect care are componentele:

   c1 = 1 şi c2 = 2

   În acest caz, se apelează în mod automat constructorul care are parametri şi care realizează iniţializarea componentelor obiectului instanţiat. Pointerul *pvb* pointează spre obiectul instanţiat în acest fel.

Operatorii *new* şi *delete* permit alocarea şi respectiv dezalocarea de tablouri în memoria *heap*. În acest caz, se utilizează construcţii de felul celor de mai jos.

Exemple:

1. `int *p = new int[10];`

   Se alocă o zonă de memorie capabilă pentru a memora 10 date de tip *int* (10*sizeof(int)).

   Adresa de început a acestei zone se atribuie lui *p*. La elementele

respective ne putem referi prin expresii de forma:

```
p[0], p[1], ... ,p[9]
```

sau

```
*p, *(p+1), ... ,*(p+9).
```

Zona respectivă se eliberează aplicînd operatorul *delete* în mod obişnuit:

```
delete p;
```

2. `char *q = new char[100];`

Se rezervă o zonă de memorie de 100 de octeţi şi adresa de început a zonei respective se atribuie lui *q*.

Dezalocarea se realizează printr-o instrucţiune de forma:

```
delete q;
```

3.

```
class vb {
        double c1,c2;
    public:
        vb()
        {c1 = 0; c2 = 0;}

        vb(double x,double y)
        {c1 = x; c2 = y;}
        ...
};
vb *ptvb;
ptvb = new vb[100];
```

Se rezervă zonă de memorie pentru 100 de obiecte de tip *vb*. În acest scop se utilizează supraîncărcarea standard a operatorului *new*. În acest caz, se apelează constructorul implicit de 100 de ori.

Adresa de început a zonei rezervate pentru cele 100 de elemente se atribuie pointerului *ptvb*.

Zona de memorie alocată în acest fel se eliberează printr-o instrucţiune de forma:

```
delete ptvb;
```

La obiectele alocate mai sus ne putem referi prin construcţii de forma:

*ptvb, *(ptvb+1), ... , *(ptvb+99).

La alocarea tablourilor cu ajutorul operatorului *new* nu se pot face inițializări. De exemplu, dacă se dorește inițializarea elementelor tabloului de mai sus în așa fel încît elementul:

*(ptvb+i)

să aibă valoarea inițială (i,i), se poate apela o funcție membru sau prieten a clasei, definită ca mai jos.

```
void vb::initab(int n)
{
 vb *t;
 t = this;

 for(i = 0; i < n;i++,t++)
     *t = vb(i,i);
}
```

Pentru a inițializa tabloul de mai sus, vom apela funcția membru astfel:

```
ptvb -> initab(100);
```

Amintim că în instrucțiunea:

```
*t = vb(i,i);
```

expresia din partea dreaptă definește un obiect *anonim* (fără nume). El se inițializează prin constructorul clasei *vb*, apoi obiectul respectiv se copiază în zona spre care pointează *t*. Obiectul anonim este distrus automat.

În rezumat, operatorul *new* poate fi apelat pentru a rezerva memorie atît pentru date de tipuri predefinite, cît și pentru tipuri abstracte. Datele alocate în acest fel pot fi inițializate.

Se pot aloca tablouri, dar în acest caz nu se mai pot face inițializări. În cazul obiectelor trebuie să precizăm următoarele:

1. Nu este obligatoriu să existe constructor definit de programator.
2. Dacă există un constructor fără parametri definit de utilizator, atunci el se aplică automat la alocări prin expresii de forma:

   new *nume_clasa*
3. Dacă se fac alocări cu inițializare:

   new *nume_clasa(par1,par2,...,parn)*

   atunci clasa trebuie să aibă un constructor care să poată realiza inițializarea respectivă.
4. Dacă se fac alocări de forma:

new *nume_clasa*[*nr_elemente*]

atunci se utilizează supraîncărcarea standard (predefinită) a operatorului *new*. Dacă clasa are un constructor implicit definit de programator, acesta se apelează la o astfel de alocare de *nr_elemente* ori.

Obiectele alocate cu ajutorul lui *new* se distrug cu ajutorul operatorului *delete*. În acest scop se utilizează o instrucţiune de forma:

```
delete p;
```

unde:

p - Este un pointer spre obiectul care se distruge.

Dacă clasa are un destructor definit de programator, atunci instrucţiunea de mai sus implică apelul destructorului respectiv.

Obiectele tablou, alocate printr-o expresie de forma:

p = new *nume_clasa*[*nr_elemente*]

se pot distruge prin una din instrucţiunile:

```
delete p;
```

sau

```
delete [nr_elemente]p;
```

Prima instrucţiune apelează în mod automat destructorul clasei o singură dată, spre deosebire de cea de a doua, care distruge obiectul apelînd de *nr_elemente* ori destructorul clasei.

Utilizatorul poate supraîncărca aceşti operatori în cazul în care supraîncărcarea standard nu este suficientă.

Supraîncărcarea operatorilor *new* şi *delete* prezintă unele diferenţe faţă de supraîncărcarea celorlalţi operatori.

O primă diferenţă constă în aceea că, supraîncărcarea operatorilor *new* şi *delete* se realizează cu ajutorul *funcţiilor membru statice*, spre deosebire de ceilalţi operatori care se supraîncarcă prin funcţii membru care nu sînt statice sau prin funcţii prieten.

Funcţiile care supraîncarcă operatorii *new* şi *delete* sînt asimilate,în mod implicit, ca funcţii membru statice, aşa că nu mai este nevoie să specificăm explicit acest lucru.

O altă particularitate a supraîncărcării acestor operatori este antetul funcţiilor prin care se realizează supraîncărcarea.

Pentru operatorul *new* se utilizează următorul prototip:

void ****operator*** **new(***size_t lung***);**

Funcţia returnează un pointer universal a cărui valoare este adresa de început a zonei de memorie rezervată în memoria *heap*.

Tipul *size_t* este definit în fişierele de bibliotecă *stddef.h* şi *stdlib.h* astfel:

typedef unsigned *size_t*;

Tipul *size_t* trebuie să fie prezent în antetul funcţiilor pentru supraîncărcarea operatorului *new* (el deşi este un alt nume pentru *unsigned*, nu poate fi înlocuit prin *unsigned*).

Antetul funcţiei pentru supraîncărcarea operatorului *new* este de forma:

void **nume_clasa::operator* **new(***size_t lung***)**

Parametrul *lung* are ca valoare dimensiunea în octeţi a zonei de memorie care se rezervă în memoria *heap*.

O altă particularitate a supraîncărcării operatorului *new* este faptul că la aplicarea lui nu se indică nici o valoare pentru parametrul formal *lung*. Aceasta, deoarece compilatorul calculează, în mod automat, dimensiunea obiectului pentru care se rezervă zonă de memorie şi această dimensiune se atribuie parametrului *lung*.

În felul acesta, operatorul *new* supraîncărcat se utilizează în mod obişnuit în expresii de forma:

new *nume_clasa*

unde:

nume_clasa - Este numele clasei pentru care este supraîncărcat.

Aplicarea operatorului *new* supraîncărcat de utilizator apelează în mod automat constructorul corespunzător clasei, dacă există constructori definiţi pentru clasa respectivă sau pe cel implicit definit de compilator dacă clasa nu are constructori. De aceea, la aplicarea operatorului *new*, pot să fie prezenţi parametri pentru constructorii clasei.

În general, operatorul *new* se aplică la un operand folosind următorul format:

new *nume_clasa(par1,par2,...,parn)*

unde:

par1,par2,...,parn - Sînt expresii şi reprezintă parametri pentru apelul unui constructor al clasei:

nume_clasa.

Supraîncărcarea predefinită (globală) a operatorului *new* poate fi apelată şi în cazul în care acesta este supraîncărcat de utilizator, folosind operatorul de rezoluţie. Astfel, o expresie de forma:

::new *nume_clasa*

aplică operatorul *new* predefinit, care, aşa cum s-a afirmat mai sus, are o supraîncărcare standard, globală.

Pentru operatorul *delete*, de asemenea există particularităţi specifice. Prototipul funcţiei pentru supraîncărcarea operatorului *delete* se poate specifica astfel:

void *operator* **delete(void** **p*);

Ca şi în cazul operatorului *new*, funcţia pentru supraîncărcarea operatorului este în mod implicit o funcţie membru *statică*.

Antetul funcţiei este de forma:

void *nume_clasa::operator* **delete (void ****p*)

Operatorul are ca operand un pointer spre obiectul care se dezalocă. Valoarea acestui pointer trebuie să fie cea definită prin aplicarea operatorului *new* la construirea obiectului.

Exemplu:

```
nume_clasa *p = new nume_clasa;
/* - se aloca zona de memorie pentru un obiect de tip nume_clasa;
   - adresa de inceput a zonei de memorie alocata se atribuie lui p */
...
delete p;  /* se elibereaza zona de memorie alocata prin new */
```

La aplicarea operatorului *delete* se apelează în mod automat destructorul clasei dacă există un destructor definit de utilizator. În caz contrar, se apelează destructorul implicit definit de compilator.

Ca şi în cazul operatorului *new*, dacă operatorul *delete* a fost supraîncărcat pentru tipul abstract *nume_clasa*, atunci operatorul *delete* predefinit (global) poate fi aplicat precedîndu-l de operatorul de rezoluţie:

::delete *nume_clasa*;

Operatorii *new* şi *delete* supraîncărcaţi de utilizator nu se folosesc la alocarea şi dezalocarea tablourilor de obiecte.

În cazul alocării şi distrugerii tablourilor de obiecte se utilizează operatorii *new* şi *delete* standard (globali).

Operatorii *new* şi *delete* definiţi de utilizator au un avantaj faţă de cei standard şi anume ei sînt mai economici din punct de vedere al timpului de execuţie şi al memoriei consumate pentru gestiunea obiectelor dinamice.

În corpul funcţiilor pentru definirea supraîncărcării operatorilor *new* şi *delete* se pot utiliza versiunile standard pentru aceşti operatori.

În încheiere amintim că, funcţia pentru supraîncărcarea operatorului *delete* are un al doilea parametru facultativ, de tip *size_t*,care este gestionat automat de compilator. El nu este obligatoriu să apară în antetul funcţiei pentru supraîncărcarea operatorului *delete*. Antetul general al funcţiei pentru supraîncărcarea operatorului *delete* este următorul:

void *nume_clasa::operator* **delete(void** **p,size_t lung*)

Exemplu:

Fie clasa de mai jos, pentru care s-au supraîncărcat operatorii *new* şi *delete*:

```
class nume_clasa {
        ...
    public:
        nume_clasa();
        void *operator new(size_t);
        void operator delete(void *);
        ...
        ~nume_clasa();
};
```

Considerăm următoarele declaraţii:

```
1.  char *pcar = new char;
2.  char *tc   = new char[100];
3.  double *td = new double[100];
4.  nume_clasa *pnume = new nume_clasa;
5.  nume_clasa *tnume = new nume_clasa[100];
```

În cazul primei declaraţii se aplică operatorul *new* standard.

Se rezervă zonă pentru un caracter (un octet) şi adresa ei se atribuie lui *pcar*. Această zonă se eliberează aplicînd operatorul *delete* standard:

```
delete pcar;
```

La declaraţia a doua se aplică operatorul *new* standard pentru a aloca o zonă de 100 de octeţi. Adresa de început a acestei zone se atribuie lui *tc*.

Zona respectivă se eliberează aplicînd operatorul delete *standard*:

```
delete tc;
```

Declaraţia a treia este analogă cu a doua, deosebirea constă în aceea că, în acest caz, se alocă 100*sizeof(double) octeţi în loc de 100 de octeţi.

La declaraţia a patra se alocă zonă pentru un obiect de tip *nume_clasa*.

La alocare se apelează constructorul implicit al clasei, definit de utilizator.

Alocarea se face apelîndu-se operatorul *new* supraîncărcat de utilizator.

Adresa de început a zonei de memorie alocată în acest fel se atribuie lui *pnume*.

Zona respectivă se eliberează aplicînd operatorul *delete* supraîncărcat definit de utilizator:

```
delete pnume;
```

Această instrucţiune apelează destructorul clasei.

La ultima declaraţie se alocă zonă pentru 100 de elemente de tip *nume_clasa*.

Se apelează operatorul *new* standard.

Adresa de început a zonei rezervate se atribuie pointerului *tnume*.

La alocare se apelează constructorul implicit de o sută de ori.

Zona respectivă se eliberează aplicînd operatorul *delete* standard:

```
delete[100] tnume;
```

sau

```
delete tnume;
```

În primul caz, se apelează destructorul de o sută de ori. În cel de al doilea caz, destructorul se apelează o singură dată.

Exerciţii:

23.11 Să se definească tipul abstract *tnod* care are următoarele date membru:

- *cuvint*: pointer spre un şir de caractere;
- *frecv*: întreg utilizat ca numărător de cuvinte;
- *urm*: pointer spre *tnod*;
- *ind*: indicator pentru citirea sfîrşitului de fişier.

Clasa care implementează tipul abstract *tnod* are funcţii membru care realizează următoarele:

- constructor;
- destructor;
- afişează datele membru *cuvint* şi *frecv*;
- returnează valoarea lui *ind*;
- supraîncarcă operatorii *new* şi *delete*;
- operatorul pentru incrementarea numărătorului *frecv*.

Clasa *slist* este clasă prieten pentru clasa *tnod*.

FIŞIERUL BXXIII11.H

```
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#ifndef __Boolean
#define __Boolean

enum Boolean {false,true};

#endif

class tnod {
    char *cuvint;
    int frecv;
    tnod *urm;
    static Boolean ind;
    friend class slist;
 public:
    tnod();
    ~tnod();
    void afistnod();
    void *operator new(size_t);
    void operator delete(void *);
    void operator++();
    static Boolean retind();
};
```

Funcţiile membru ale clasei *tnod* se definesc în fişierul de extensie *CPP*, de mai jos.

Indicatorul *ind* are valoarea *false* atît timp cît nu s-a întîlnit sfîrşitul de fişier.

FIŞIERUL BXXIII11

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __STRING_H
#include <string.h>
#define __STRING_H
#endif
#ifndef __BXXIII11_H
#include "BXXIII11.H"
#define __BXXIII11_H
#endif

// Initializarea variabilei ind

Boolean tnod::ind = false;

tnod::tnod()
/* - citeste cuvintul curent daca ind = false;
   - pastreaza cuvintul citit in memoria heap;
   - realizeaza initializarile:
        frecv = 1;
        urm = 0.
*/
{
 char t[255];
 int sf;
 cuvint = 0;

 if(tnod::ind == false){
     while((sf = scanf("%s",t))!=EOF){
         char *p = t;
         int c;

          // salt peste caractere care nu sint litere
         while((c =* p)&&(c < 'A' || c > 'Z' &&
                c <'a' || c > 'z'))
           p++;
         if(c == 0)    // nu sint litere
```

```
            continue;

        // pastreaza inceputul cuvintului
        char *q = p;

        // cauta sfirsitul cuvintului
        while((c=*p)&&(c >= 'A'&&c <= 'Z'||
              c >= 'a'&&c <= 'z'))
            p++;
        *p = '\0';    // caracterul NUL la sfirsitul cuvintului

        // rezerva zona pentru cuvint in memoria heap
        // se apeleaza operatorul new standard

        printf("tnod::tnod %lu\n",coreleft());
        if((cuvint = new char[strlen(q)+1]) == 0){
            printf("memorie insuficienta\n");
            exit(1);
        }
        printf("tnod::tnod %lu\n",coreleft());

        // se transfera cuvintul in zona rezervata
        strcpy(cuvint,q);

        // intializari

        frecv = 1;
        urm = 0;    // pointerul nul
        break;
    } // sfirsit while

    if(sf == EOF)   // s-a intilnit EOF
        tnod::ind = true;
  }
} // sfirsit constructor

inline tnod::~tnod()
{
 // se apeleaza operatorul delete standard
 printf("\ndestructor tnod %lu\n",coreleft());
 delete cuvint;
 printf("\ndestructor tnod %lu\n",coreleft());
}
```

```
inline void tnod::afistnod()
// afiseaza cuvintul si frecventa
{
 printf("cuvintul=%s\t are frecventa=%d\n",
        cuvint,frecv);
}

void *tnod::operator new(size_t lung)
// supraincarcarea operatorului new
{
 tnod *p;

 // se aplica operatorul new standard
 printf(" operator new %lu\n",coreleft());
 p=(tnod *)new char[lung];
 printf(" operator new %lu\n",coreleft());
 return p;
}

void tnod::operator delete(void *p)
// supraincarcarea operatorului delete
{
// se aplica operatorul delete standard
 printf(" operator delete %lu\n",coreleft());
 ::delete p;
 printf(" operator delete %lu\n",coreleft());
}

void tnod::operator++()   // incrementeaza frecv
{
 frecv++;
}

Boolean tnod::retind()   // returneaza valoarea ind
{
 return tnod::ind;
}
```

Observaţie:

Funcţia *coreleft* returnează dimensiunea, în octeţi, a memoriei libere în momentul apelului. Ea a fost apelată în mai multe funcţii pentru a putea urmări momentele în care se alocă şi dezalocă obiectele care sînt instanţieri ale clasei *tnod*.

23.12 Să se definească tipul abstract *slist* pentru implementarea listei simplu înlănţuite.

Listele simplu înlănţuite au fost implementate în limbajul C în capitolul 11. Pentru gestiunea listelor simplu înlănţuite s-au folosit două variabile *prim* şi *ultim* care sînt pointeri spre primul şi respectiv ultimul nod al listei.

Aceste valori se vor utiliza şi în cazul de faţă cu acelaşi scop. Ele sînt date membru protejate ale tipului abstract *slist*.

Avantajul implementării tipului abstract *slist* este acela că, se pot defini simultan şi simplu oricîte obiecte de tip listă.

Funcţiile membru ale tipului abstract *slist* sînt:

- constructor implicit;
- destructor;
- adăugarea unui nod după ultimul nod al listei;
- inserarea unui nod înaintea primului nod al listei;
- ştergerea primului nod al listei;
- ştergerea ultimului nod al listei;
- căutarea în listă a unui nod;
- afişarea datelor aflate în nodurile listei.

Nodurile listei sînt obiecte de tip *tnod*, tip definit în exerciţiul 23.11.

FIŞIERUL BXXIII12.H

```
#ifndef __BXXIII11_H
#include "BXXIII11.CPP"
#define __BXXIII11_H
#endif

class slist {
    tnod *prim;
    tnod *ultim;
 public:
    slist();     //constructor
    ~slist();    //destructor
    tnod *adauga(tnod *);    //adauga un nod dupa cel spre
                             //care pointeaza ultim
    tnod *insereaza(tnod *p); //insereaza un nod inaintea
                              //celui spre care pointeaza
```

```
                                                    // prim
        void sprim();     // sterge primul nod din lista
        void sultim();  // sterge ultimul nod al listei

        tnod *cauta(tnod *p);
        /* - cauta nodul din lista pentru care p -> cuvint pointeaza
             spre un sir identic cu cel spre care pointeaza cuvint
             din nod;
           - returneaza pointerul spre nodul respectiv sau zero daca
             nu exista un astfel de nod */

        void afislist(); // afiseaza datele din nodurile listei
};
```

Funcţiile membru ale clasei *slist* se definesc în fişierul de extensie *CPP*, de mai jos.

FIŞIERUL BXXIII12

```
#ifndef __BXXIII12_H
#include "BXXIII12.H"
#define __BXXIII12_H
#endif

inline slist::slist() // constructor
{
 prim = ultim = 0;
}

slist::~slist()     // destructor; distruge toate nodurile listei
{
 tnod *p,*q;

 for(p=prim;p;p=q){
     q=p->urm;    // pointer spre nodul urmator
     delete p;    // operatorul delete supraincarcat
 }
 prim = ultim = 0;
}

tnod *slist::adauga(tnod *p)
// adauga un nod dupa ultimul nod al listei
{
 if(prim == 0)   // lista vida
```

```
    prim = p;
  else
    ultim -> urm = p;
  ultim = p;
  return p;
}

tnod *slist::insereaza(tnod *p)
// insereaza un nod inaintea celui spre care pointeaza prim
{
  if(prim==0)  // lista vida
    ultim = p;
  else
    p -> urm = prim;
  prim = p;
  return p;
}

void slist::sprim()  // sterge primul nod din lista
{
  if(prim){
    tnod *p = prim -> urm;
    delete prim; // se aplica operatorul delete supraincarcat
    prim = p;
    if(prim == 0)
      ultim = 0;  // lista a devenit vida
  }else
    printf("lista vida\n");
}

void slist::sultim()  // sterge ultimul nod al listei
{
  if(prim){
    tnod *p,*pl;

    // p - pointeaza spre nodul curent
    p = prim;

    // p1 - pointeaza spre nodul precedent
    pl = 0;
    while(p != ultim) {  // parcurge lista
      pl = p;
      p = p -> urm;
```

```
        }
        // p - pointeaza spre ultimul nod;
        // p1 - pointeaza spre nodul precedent
        if(p1 == 0){   // lista are un singur nod si devine vida
            delete p;   // se aplica operatorul supraincarcat
            prim = ultim = 0;
            return;
        }

        // nodul spre care pointeaza p1 devine ultimul nod al listei
        p1 -> urm = 0;
        ultim = p1;

        // se sterge nodul spre care pointeaza p
        delete p;   // se aplica operatorul supraincarcat
    }else
        printf("lista vida\n");
}

tnod *slist::cauta(tnod *p)
/* - cauta nodul pentru care cuvint pointeaza spre un sir identic cu cel
     spre care pointeaza p -> cuvint;
   - returneaza pointerul spre nodul respectiv sau 0 daca nu exista un
     astfel de nod. */
{
 tnod *q;

 for(q=prim;q;q=q->urm)
     if(stricmp( p -> cuvint, q -> cuvint) == 0)
         return q;
 return 0;
}

void slist::afislist()
/* - afiseaza pentru fiecare nod:
      - cuvint
    si
      - frecv.
*/
{
 int nrlin = 1;

 for(tnod *p = prim; p; p = p -> urm){
```

```
        p -> afistnod();
        nrlinn++;
        if(nrlin%23==0){
            printf("Actionati o tasta pentru a\
                    continua\n");
            getch();
        }
    }
}
```

23.13 Să se scrie un program care citeşte un text şi afişează frecvenţa de apariţie a fiecărui cuvînt din text. Nu se face distincţie între literele mari şi mici.

Această problemă a fost rezolvată în mai multe moduri în capitolele precedente.

În particular, în capitolul 11 se dă o rezolvare folosind listele simplu înlănţuite.

Programul de faţă utilizează aceeaşi metodă.

PROGRAMUL BXXIII13

```
#include <alloc.h>
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#include "BXXIII12.CPP"

main()
/* citeste un text si afiseaza frecventa cuvintelor citite */
{
 slist lista;
 tnod *nod,*p;

 printf("%lu\n",coreleft());
 while(tnod::retind() == false)
 // nu s-a ajuns la sfirsitul textului
 {
     nod = new tnod;    // se aplica operatorul new supraincarcat
     // cauta nodul in lista
     if(tnod::retind()==false)
```

```
        if(p = lista.cauta(nod)){
        // cuvintul exista deja in lista
            (*p)++;    // incrementeaza frecv
            delete nod;      // se aplica operatorul delete
                             // supraincarcat
        }else
            lista.adauga(nod);
            /* nodul spre care pointeaza nod contine un cuvint
               negasit in lista; se adauga la lista */
    else
        delete nod;
    printf("Pentru a continua actionati o\
            tasta\n");
    getch();
  } // sfirsit while

 // listeaza nodurile listei
 lista.afislist();
}
```

23.2. Supraîncărcarea operatorului =

În limbajul C++ se pot face atribuiri de obiecte care sînt instanţieri ale aceleeaşi clase.

Exemplu:

Fie instanţierile:

nume_clasa obiect1,obiect2;

...

O atribuire de forma:

obiect2 = obiect1;

este acceptată de compilator.

La atribuirea de obiecte, în mod implicit, se atribuie datele membru ale obiectului din dreapta operatorului = la datele membru corespunzătoare ale obiectului din stînga aceluiaşi operator. Astfel de atribuiri se pot utiliza şi în declaraţii.

Exemplu:

```
class complex {
    double real;
```

```
        double imag;
    public:
        complex(double x=0,double y=0)
        {
            real = x;
            imag = y;
        }
        ...
};

complex z(1,2);     //z = 1+2i
complex z1 = z;     //z1 = 1+2i
complex z2;
...
z2 = z1;            //z2 = 1+2i
```

Atît în cazul declaraţiei lui *z1*, cît şi în cazul instrucţiunii de atribuire pentru *z2*, se copiază componentele date ale obiectului complex din dreapta caracterului "=" în zona de memorie alocată componentelor date ale obiectului aflat în stînga aceluiaşi caracter.

În cazul declaraţiei lui *z1*, caracterul "=" se utilizează pentru a realiza o iniţializare, iar în cazul instrucţiunii:

```
z2 = z1;
```

acelaşi caracter se utilizează pentru a realiza o atribuire.

Ambele operaţii sînt definite în mod implicit şi se realizează aşa cum s-a indicat mai sus, adică prin simpla copiere a datelor membru.

B. Stroustrup numeşte "bitwise copy" (copiere la nivel de biţi) copierile de acest fel.

În cazul obiectelor clasei *complex*, copierea din cadrul operaţiei de iniţializare este identică cu cea utilizată în cadrul operaţiei de atribuire.

În general, cele două operaţii nu sînt identice. De asemenea, copierea de tip "bitwise copy" nu este totdeauna suficientă. Acest fapt rezultă imediat dacă reluăm exemplul pentru implementarea tipului abstract *string* (vezi 22.9) sau *sir* (vezi exerciţiul 22.12.).

Considerăm clasa *sir*, definită ca mai jos:

```
class sir {
    char *psir;
    int lung;
```

```
public:
    sir(char *s);
    sir(int nrcar = 70);
    int retlung();      //returneaza lungimea sirului
    void afsir();       //afiseaza sirul definit de psir
    int citsir();       //citeste un sir de caractere
    ~sir() {delete psir;}
};
```

Funcţiile membru de mai sus, sînt definite în exerciţiul 22.12.

Fie declaraţia:

```
sir sir1("C++ este un C mai bun");
```

La instanţierea obiectului *sir1* se apelează constructorul:

```
sir(char *s)
```

Acesta rezervă zonă de memorie în memoria *heap* pentru textul:

C++ este un C mai bun

şi atribuie adresa de început a acestei zone pointerului:

```
sir1.psir
```

De asemenea, se atribuie datei membru:

```
sir1.lung
```

valoarea 21 (numărul de caractere din compunerea textului de iniţializare a obiectului sir1, fără a considera şi caracterul NUL).

Să considerăm declaraţia pentru instanţierea obiectului *sir2* cu iniţializarea acestuia folosind obiectul *sir1*:

(1) `sir sir2 = sir1;`

Utilizatorul foloseşte o astfel de declaraţie pentru a obţine acelaşi efect ca şi cînd ar utiliza declaraţia:

(2) `sir sir2("C++ este un C mai bun");`

În realitate, cele două declaraţii nu realizează acelaşi lucru.

În cazul declaraţiei (1), la instanţierea lui *sir2* se apelează constructorul:

```
sir(int nrcar = 70)
```

care rezervă o zonă de memorie de 70 de octeţi în care se păstrează sirul vid.

Adresa de început a acestei zone se atribuie lui *sir2.psir*, apoi se copiază

bit cu bit (bitwise copy) valorile componentelor obiectului *sir1* (*sir1.psir* şi *sir1.lung*) în zonele de memorie alocate componentelor corespunzătoare ale obiectului *sir2*. În felul acesta, *sir2.psir* are aceeaşi valoare ca şi *sir1.psir*. Zona de memorie de70 de octeţi alocată în memoria heap este pierdută şi nu se mai poate face acces la ea. De asemenea, textul de iniţializare este păstrat într-un singur exemplar, situaţie care, de obicei, va conduce la erori în prelucrarea ulterioară a obiectelor *sir1* şi *sir2*.

De asemenea, la distrugerea obiectelor *sir1* şi *sir2* ar urma ca textul de iniţializare, păstrat într-un singur exemplar, să fie eliberat din memoria *heap* de două ori. De aceea, copierea implicită bit cu bit a obiectelor nu este o soluţie corectă pentru obiectele clasei *sir*.

Instanţierea lui *sir2* cu ajutorul declaraţiei (2) nu are nici o legătură cu instanţierea lui *sir1* şi se rezervă o zonă de memorie în memoria *heap* distinctă faţă de cea rezervată pentru acelaşi text la instanţierea lui *sir1*. De aceea,

```
sir1.psir
```

şi

```
sir2.psir
```

au valori diferite.

Acelaşi lucru este valabil şi în cazul atribuirilor. Fie de exemplu instanţierile:

```
sir sir1("C++ este un C mai bun");
sir sir2;
```

Atribuirea:

```
sir2 = sir1;
```

realizează o copiere bit cu bit, deci *sir1.psir* şi *sir2.psir* devin egale şi pointează spre aceeaşi zonă de memorie din memoria *heap* care a fost rezervată la instanţierea lui *sir1*. Ca şi în cazul declaraţiei (2), zona de memorie din memoria *heap* alocată la instanţierea lui *sir2* este pierdută în urma atribuirii, deoarece ea devine inaccesibilă.

Din cele de mai sus, rezultă că pentru obiectele care sînt instanţieri ale clasei *sir* nu este suficientă copierea implicită bit cu bit. În astfel de cazuri este nevoie ca utilizatorul să definească copieri specifice, care să ţină seama de natura componentelor obiectelor.

În general, operaţia de iniţializare prin copiere diferă de cea de atribuire, deoarece iniţializarea este însoţită de alocare. De aceea, sînt necesare două tipuri de copieri, una care să se realizeze la iniţializare prin copiere, adică la

declaraţii de forma:

(3) `nume_clasa obiect2 = obiect1;`

şi una pentru atribuiri de obiecte:

(4) `obiect2 = obiect1;`

Iniţializarea prin copiere se realizează cu ajutorul *constructorului de copiere* care a fost definit în paragraful 22.8.

Un astfel de constructor are un prototip de forma:

nume_clasa(**const** *nume_clasa* & *obiect*);

Menţionăm că, un astfel de constructor se apelează automat la instanţierea unui obiect printr-o declaraţie de forma (3).

Copierea bit cu bit se realizează numai dacă clasa nu are un astfel de constructor.

Clasa *sir*, definită în exerciţiul 22.12., are un constructor de copiere definit ca mai jos:

```
sir::sir(const sir& s)
{
 lung = s.lung;    // se copiaza lungimea sirului

 // se aloca zona pentru sir
 psir = new char[lung+1];

 // se copiaza sirul in zona rezervata
 strcpy(psir, s.psir);
}
```

Acest constructor se va apela automat la întîlnirea declaraţiei (1) şi rezultatul lui constă în aceea că declaraţia (1) are acelaşi efect ca şi declaraţia (2).

Constructorul de copiere nu se apelează la atribuiri.

Pentru a soluţiona în mod corect atribuirile de obiecte de felul celor de mai sus, este necesar să supraîncărcăm în mod corespunzător operatorul de atribuire (=).

În principiu, constructorii de copiere şi supraîncărcarea operatorului de atribuire sînt necesari pentru clasele care au ca date membru pointeri spre zone alocate în memoria *heap* (alocate dinamic).

Un alt caz în care este nevoie de copierea obiectelor este acela cînd se apelează funcţii care au ca parametri obiecte sau care returnează obiecte.

În astfel de cazuri se alocă obiecte temporare pe stivă şi de aceea intervin operaţii similare cu cele de la iniţializare.

Pentru ca aceste transferuri de obiecte să se facă corect pentru obiecte cu componente pointeri spre date din memoria *heap*, este necesar să avem constructor de copiere în clasa respectivă.

Acesta se va apela automat atît la transferul prin parametri a obiectelor, cît şi în cazul în care funcţia returnează un obiect.

În absenţa constructorului de copiere, transferul prin parametri a obiectelor precum şi returnarea de obiecte, se realizează aplicîndu-se copierea implicită, adică copierea bit cu bit.

Un exemplu în care se utilizează copierea bit cu bit, la returnarea de obiecte, este clasa *complex*, implementată în exerciţiul 22.1.

În legătură cu supraîncărcarea operatorului de atribuire este necesar să amintim următoarele:

- supraîncărcarea operatorului de atribuire se poate realiza numai printr-o funcţie membru care nu este statică;
- obiectele operatorului se pot transfera prin valoare sau referinţă;
- obiectul din stînga operatorului de atribuire, dacă are componente pointeri spre zone alocate în memoria *heap*, trebuiesc eliberate înainte de a se aloca zone noi în vederea copierii elementelor corespunzătoare ale obiectului din dreapta operatorului respectiv; se obişnuieşte să se spună că obiectul din stînga operatorului de atribuire trebuie "curăţat" înainte de a se atribui valorile obiectului din dreapta operatorului de atribuire.

De obicei, funcţia pentru supraîncărcarea operatorului de atribuire are un parametru de tip referinţă şi returnează o referinţă la obiectul curent. Aceasta deoarece transferul prin referinţă este mai eficient decît transferul obiectelor.

Avînd în vedere acest fapt, se recomandă următorul antet pentru supraîncărcarea operatorului de atribuire:

nume_clasa& nume_clasa::operator = (**const** *nume_clasa & operand_drept*)

Obiectul curent este operandul stîng al atribuirii.

O astfel de funcţie se apelează automat pentru realizarea atribuirilor de forma:

obiect2 = obiect1;

unde:

obiect1 şi obiect2 - Sînt instanţieri ale clasei *nume_clasa*.

Atribuirile de forma:

```
obiect1 = obiect1;
```

nu au nici un efect. De aceea, atribuirile de acest fel se vor elimina. În acest scop, în corpul funcţiei pentru supraîncărcarea operatorului de atribuire se testează dacă operatorul curent este diferit de cel referit prin parametru şi numai în acest caz se execută corpul funcţiei.

Folosind antetul de mai sus, un astfel de test se poate realiza astfel:

```
if(this != &operand_drept){
// operanzi diferiti
  ...
}else    // operanzi identici
 return *this;
```

Operatorul de atribuire are şi o altă formă care se utilizează pentru a face prescurtări. Este vorba de formatul:

op=

unde:

op - Este un operator binar aritmetic sau logic pe biţi.

Aceste variante pot fi şi ele supraîncărcate în mod corespunzător. Menţionăm că, dacă operatorii *op* şi = sînt supraîncărcaţi, nu decurge că este supraîncărcat şi operatorul op=. Acesta trebuie supraîncărcat separat în mod corespunzător.

Operatorul op= se supraîncarcă folosind funcţii membru nestatice care au un prototip similar cu funcţiile pentru supraîncărcarea operatorului de atribuire.

Exerciţii:

23.14 Să se implementeze tipul abstract *sir* pentru instanţierea şirurilor de caractere.

Tipul *sir* a fost definit în exerciţiul 22.12.

În exerciţiul de faţă, se schimbă funcţia membru *atribsir* a clasei *sir* cu funcţia care supraîncarcă operatorul de atribuire.

În plus, se supraîncarcă operatorii + şi += pentru concatenarea şirurilor de caractere.

Expresia:

```
sir1 = sir2+sir3
```

este legală pentru obiectele clasei *sir* şi ea atribuie lui *sir1*, şirul obţinut prin concatenarea lui *sir3* la sfîrşitul lui *sir2*.

Expresia:

```
sir1 += sir2
```

este legală pentru obiectele clasei *sir* şi ea are ca efect concatenarea lui *sir2* la sfîrşitul lui *sir1*.

FIŞIERUL BXXIII14.H

```
class sir {
    char *psir;
    int lung;
  public:
    sir(char *s); /* constructor pentru initializarea obiectului
                     cu pointerul spre copia in memoria heap a
                     sirului spre care pointeaza s */
    sir(int nrcar=70); /* constructor care rezerva zona de
                          memorie in memoria heap si pastrea-
                          za in ea sirul vid */
    sir( const sir&); /* constructor de copiere */
    ~sir();    // destructor
    int retlung();   // returneaza lungimea sirului
    void afsir();    // afiseaza sirul
    int citsir();    // citeste un sir de caractere
    sir& operator = (const sir&); // supraincarca =
    sir operator+(sir&);   /* returneaza un obiect de tip
                              sir obtinut prin concatenarea
                              parametrului la sfirsitul
                              obiectului curent */
    sir& operator += (sir&); /* concateneaza parametrul la
                                sfirsitul obiectului curent */
};
```

Fişierul de mai jos este de extensie .cpp şi el conţine definiţiile funcţiilor membru.

FIŞIERUL BXXIII14

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
```

```
#endif
#ifndef __STRING_H
#include <string.h>
#define __STRING_H
#endif
#ifndef __SIR_H
#include "BXXIII14.H"
#define __SIR_H
#endif

sir::sir(char *s)
/* constructor utilizat la initializarea obiectului cu pointerul spre copia
   in memoria heap a sirului spre care pointeaza  s */
{
 lung = strlen(s);   // lungimea sirului
 psir=new char[lung+1];   // rezerva zona in memoria heap
 strcpy(psir,s);   // transfera sirul in memoria heap
}

sir::sir(int dim)
/* - constructor utilizat pentru a rezerva in memoria heap o zona de
     dim+1 octeti;
   - pastreaza sirul vid in zona respectiva. */
{
 lung = dim;   // pastreaza lungimea
 psir = new char[lung+1];   // rezerva zona
 *psir = '\0';
}

sir::sir(const sir& s)   // constructor de copiere
{
 lung = s.lung;   // pastreaza lungimea sirului
 psir = new char[lung+1];   // rezerva zona
 strcpy(psir,s.psir);   // copiaza sirul
}

inline sir::~sir()   // destructor
{
 delete psir;
}
```

```
inline int sir::retlung()
/* returneaza numarul de caractere din compunerea sirului fara
   caracterul NUL */
{
 return lung;
}

inline void sir::afsir()
// afiseaza sirul spre care pointeaza psir
{
 printf(psir);
 printf("\n");
}

int sir::citsir()
/* - citeste un sir de la intrarea standard si-l pastreaza in zona heap
     rezervata pentru obiectul curent;
   - returneaza:
        0 - la sfirsit de fisier;
        -1 - la trunchiere;
        1 - altfel.
*/
{
 char temp[255];

 if(gets(temp) == 0)
     return 0;   // s-a intilnit EOF
 strncpy(psir,temp,lung);  /* se copiaza sirul citit */
 *(psir+lung) = '\0';
 if(strlen(temp) > lung)
     return -1;  // trunchiere
 else
     return 1;
}

sir& sir::operator = (const sir & operand_drept)
// supraincarca operatorul =
{
 if(this != &operand_drept){
 // operanzi diferiti
     delete psir;  // curata obiectul curent
     lung = operand_drept.lung;    // pastreaza lungimea
     psir = new char[lung+1];   // rezerva zona
```

```
        // transfera sirul in zona rezervata
        strncpy(psir,operand_drept.psir,lung);
         *(psir+lung) = '\0';
   }
   return *this;
  }

 sir sir::operator+(sir& sir2)
 /* returneaza un obiect care este rezultatul concatenarii obiectului
    referit de sir2 la obiectul curent */
 {
  sir rtemp;

  delete rtemp.psir;    // curata obiectul rtemp
  rtemp.lung = lung+sir2.lung;  /* determina lungimea sirului
                                   rezultat */

  // rezerva memorie pentru sirul rezultat
  rtemp.psir = new char[rtemp.lung+1];
  strcpy(rtemp.psir,psir);    // transfera sirul curent

  /* concateneaza sirul referit de sir2 la sfirsitul sirului spre care
     pointeaza rtemp.psir */
  strcat(rtemp.psir,sir2.psir);
  return rtemp;
 }

 sir& sir::operator += (sir& operand_drept)
 /* concatenarea obiectului operand_drept la obiectul curent */
 {
  /* pentru concatenare se utilizeaza operatorul + supraincarcat,
     iar pentru atribuire operatorul = supraincarcat */

  *this = *this+operand_drept;
  return *this;
 }
```

23.15 Să se scrie un program care citeşte întregi din intervalul [1,99] şi afişează exprimările lor prin cuvinte.

Programul construieşte exprimarea în cuvinte a numerelor din intervalul respectiv prin concatenarea componentelor sale.

Exemple:

11 unsprezece
12 doisprezece
...
15 cincisprezece
16 saisprezece
17 saptesprezece
...
20 douazeci
21 douazeci si unu
22 douazeci si doi
...
29 douazeci si noua
30 trezeci
...
37 treizeci si sapte
...
40 patruzeci
...
45 patruzeci si cinci
...
60 saizeci
...
66 saizeci si sase
...
70 saptezeci
...
76 saptezeci si sase
...
99 nouazeci si noua

Se observă că numerele peste 10, dar mai mici decît 20 se termină în sufixul *sprezece*. De asemenea, dacă considerăm numerele de la 1 la 9, observăm că exprimarea în cuvinte a acestora se utilizează drept componente pentru exprimarea în cuvinte a numerelor mai mari. Există cîteva excepţii, ca de exemplu:

11, 16, 20-29 etc.

Ţinînd seama de aceste observaţii, vom proceda ca mai jos.

Se defineşte tabloul cu denumirile unităţilor:

```
char *unitate[] = {
        "",
        "unu",
        "doi",
        ...
        "noua"
};
```

Pentru un număr *n* din intervalul [1,9] exprimarea este dată de elementul *unitate[n]*.

Pentru un număr *n* din intervalul [12,19] diferit de 16, exprimarea se obţine concatenînd *unitate[n/10]* cu *sprezece*.

Pentru numerele *n* mai mari decît 20, exprimările în cuvinte se termină prin:

`si unitate[r],`

unde r = n%10 şi *n* nu este multiplu de 10.

Începutul exprimării în cuvinte se defineşte ţinînd seama de cifra zecilor:

- *douazeci* pentru 20-29;
- *treizeci* pentru 30-39;
- *patruzeci* pentru 40-49;
- *cincizeci* pentru 50-59;
- *saizeci* pentru 60-69;
- *saptezeci* pentru 70-79;
- *optzeci* pentru 80-89;
- *nouazeci* pentru 90-99.

Aceste cuvinte au sufixul *zeci*, iar prefixul este un element al tabloului *unitate*, exceptînd intervalele 20-29 şi 60-69. Pentru primul interval prefixul este *doua*, iar pentru al doilea interval prefixul este *sai*.

Rezultă următorul procedeu:

- dacă n/10=2, atunci prefixul *doua* se concateneaza cu *zeci*;
- dacă n/10=6, atunci prefixul *sai* se concatenează cu *zeci*;
- altfel `unitate[n/10]` se concatenează cu *zeci*.

PROGRAMUL BXXIII15

```
#include <stdlib.h>
#include "BXXIII14.CPP"  // clasa sir
#include "BVIII2.CPP"    // pcit_int
#include "BVIII3.CPP"    // pcit_int_lim

main()
/* - citeste intregi zecimali din intervalul [1,99];
   - afiseaza exprimarile lor in cuvinte
*/
{
 int n;
 char *unit [] = {"", "unu", "doi", "trei",
                  "patru", "cinci", "sase",
                  "sapte", "opt", "noua" };
 sir zece("zece");
 sir spre("spre");
 sir un("un");
 sir sai("sai");
 sir doua("doua");
 sir zeci("zeci ");
 sir si("si ");
 sir rez;
 int q,r;

 for(;;) {
    if(pcit_int_lim("n = ",1,99,&n) == 0)
       exit(0);
    q = n/10;
    r = n%10;
    switch(q){
       case 0: {     //[1,9]
          sir rez0(unit [n]);
          rez = rez0;
          break;
       }
       case 1: {      //[10,19]
          if(n == 10)
             rez = zece;
          else
             if(n == 11)
                rez = un+spre+zece;
             else
```

```
                if(n == 16)
                    rez = sai+spre+zece;
                else {
                    sir rez0(unit[r]);
                    rez = rez0+spre+zece;
                }
            break;
        }
        case 2: {       // [20-29]
            rez = doua+zeci;
            if(n == 20)
                break;
            rez += si;
            sir rez0(unit[r]);
            rez += rez0;
            break;
        }
        case 6: {       // [60-69]
            rez = sai+zeci;
            if(n == 60)
                break;
            rez += si;
            sir rez0(unit[r]);
            rez += rez0;
            break;
        }
        default: {      // [30-39], [40-49], [50-59],
                        // [70-79], [80-89], [90-99]
            sir rez0(unit[q]);
            rez = rez0+zeci;
            if(r == 0)
                break;
            rez += si;
            sir rez1(unit[r]);
            rez += rez1;
        }
    } // sfirsit switch

    // afisare n si exprimarea prin cuvinte
    printf("\n n = %d\t",n);
    rez.afsir();
  }
}
```

Observaţie:

Corpul funcţiei poate fi scris mai compact folosind instrucţiuni *if* în locul instrucţiunii *switch*:

```
if(q == 0){  //[1,9]
 sir rez0(unit[n]);
 rez = rez0;
}
else
 if (q == 1) { //[10,19]
   if(n == 10)
       rez = zece;
   else {
     if(n==11)
         rez = un;
     else
         if(n==16)
             rez = sai;
         else {
             sir rez0 (unit[r]);
             rez = rez0;
         }
     rez += spre + zece;
   }
} //sfirsit [10,19]
else{
   if(q==2) //[20,29]
     rez = doua;
   else
     if(q==6) //[60,69]
       rez = sai;
     else{
       sir rez0(unit[q]);
       rez=rez0;
     }
   rez += zeci;
   if(r){
     sir rez0(unit[r])
     rez += si+rez0;
   }
 }
```

23.3. Supraîncărcarea operatorului [] (operatorul de indexare)

Operatorul [] predefinit se utilizează pentru a face acces la elementele unui tablou.

De exemplu, dacă *tab* este un tablou unidimensional de un tip predefinit, atunci o expresie de forma:

(1) `tab[exp]`

permite acces la elementul tabloului *tab* de indice *exp,exp* fiind o expresie care furnizează o valoare de tip întreg.

Construcţia(1) a fost numită *variabilă cu indici.* Ea este o expresie formată din doi operanzi:

tab şi *exp*

la care se aplică operatorul de indexare ([]). De obicei, expresia (1) are şi o interpretare unitară şi anume se consideră ca formînd un *operand.* Această interpretare este posibilă datorită faptului că parantezele pătrate sînt operatori de prioritate maximă.

Interpretarea expresiei (1) ca operand se justifică datorită faptului că ea are utilizări similare cu numele unei variabile simple: ambele permit acces la o dată de un tip predefinit.

Construcţia de mai sus este valabilă şi în cazuri mai generale, cînd primul operand este chiar o expresie de pointeri, care are ca valoare un pointer.

Exemplu:

```
int *p;
int i,j;
```

Expresiile de mai jos sînt legale:

```
p[1]
p[i]
p[i+2]
p[i-1]
(p+2)[i*3+j]
(p-10)[i*3-2*j]
```

Utilizatorul poate supraîncărca operatorul [] pentru a da sens construcţiilor de felul celor de mai sus şi în cazul în care operanzii sînt obiecte.

Expresiile de forma (1) sînt expresii *lvalue* (se pot utiliza ca parte stîngă într-o atribuire). De aceea,la supraîncărcarea operatorului [] pentru tipuri

abstracte, ca şi la supraîncărcarea operatorului =, se va utiliza o funcţie membru nestatică, funcţie care să returneze o referinţă la elementul selectat prin funcţia respectivă.

În aceste condiţii, funcţia pentru supraîncărcarea operatorului [] are antetul general:

tip& nume_clasa::operator[](tip_indice i)

Odată supraîncărcat operatorul [] ca mai sus, sînt legale expresiile de forma:

(2) *obiect*[*expresie*]

unde:

obiect - Este o instaţiere a clasei *nume-clasa*.

expresie - Este o expresie care are tipul *tip_indice* sau un tip convertibil spre acesta.

Expresia (2) are ca rezultat o referinţă la elementul definit prin funcţia care supraîncarcă operatorul []. Ea este o *lvalue* şi deci se poate utiliza în ambele părţi ale unei expresii de atribuire. De asemenea, expresia (2) se poate interpreta ca fiind un operand, neglijînd faptul că ea se compune din doi operanzi la care se aplică operatorul de indexare.

Operandul de această formă reprezintă un apel al funcţiei membru operator []. Astfel, operandul (2) de mai sus, se evaluează prin apelul:

(3) *obiect.operator[] (expresie)*

unde:

operator [] - Reprezintă numele funcţiei membru care supraîncarcă operatorul [] şi ea este apelată pentru obiectul *obiect* şi indicele *expresie*.

Interesul pentru formatul (2) faţă de apelul exprimat prin (3) rezultă din faptul că reprezintă o exprimare clară şi care este încetăţenită pentru tipurile predefinite de date.

Exerciţii:

23.16 Să se definească tipul abstract *nzl* care gestionează numărul zilelor din lunile calendaristice.

Obiectele clasei *nzl* se pot utiliza în expresii de forma:

```
obiect[n]
```

sau

```
obiect[den_luna]
```

unde:

n - Este numărul lunii calendaristice.
den_luna - Este un pointer spre denumirea unei luni calendaristice.

Aceste expresii au ca valoare numărul zilelor corespunzătoare unei luni calendaristice. Acest număr de zile, este în mod implicit, numărul zilelor dintr-o lună calendaristică dar el poate fi mai mic, exprimînd numărul zilelor lucrătoare sau lucrate de o persoană în luna respectivă etc.

FIŞIERUL BXXIII16.H

```
class nzl {
      int tnz[13]; /* numarul curent de zile din lunile
                              calendaristice */
      static int tnrz[13]; /* tabloul cu numarul de zile
                              din lunile calendaristice */
      static char *tdenluna[13];
      /* tablou de pointeri spre denumirile lunilor calendaristice */
  public:
      int& operator[] (char *dl);
      /* - returneaza o referinta la elementul tnz[nl];
         - nl este numarul lunii spre a carui denumire pointeaza dl */
      int& operator[](int nl); /* returneaza o referinta
                                  la elementul tnz[nl] */
      nzl(int an); /* elementele tnz se initializeaza cu
                              cele ale lui tnrz */
      static char *retdenl(int n);
      /* returneaza denumirea lunii a n-a */
};
```

Mai jos, se iniţializează datele membru statice şi se definesc funcţiile membru.

FIŞIERUL BXXIII16

```
#ifndef __NZL_H
#include "BXXIII16.H"
#define __NZL_H
#endif

int nzl::tnrz[] = {0,31,28,31,30,31,30,31,
                    31,30,31,30,31};
```

```
char *nzl::tdenluna[] = {
 "",
 "ianuarie",
 "februarie",
 "martie",
 "aprilie",
 "mai",
 "iunie",
 "iulie",
 "august",
 "septembrie",
 "octombrie",
 "noiembrie",
 "decembrie"
};

int& nzl::operator[](char *dl)
/* - returneaza o referinta la elementul tnz[nl];
   - nl este numarul lunii spre a carei denumire pointeaza dl;
   - returneaza referinta spre zero daca nu exista o luna cu denumirea
     spre care pointeaza dl */
{
 for(int nl=1;nl < 13;nl++)
     if(stricmp(nzl::tdenluna[nl],dl) == 0)
          return tnz[nl];
 return tnrz[0];   // returneaza referinta la zero
}

int& nzl::operator[](int nl)
/* returneaza o referinta la tnz[nl] sau la zero daca nl nu se afla in
   intervalul [1,12] */
{
 return nl < 1||nl >12 ? tnrz[0]:tnz[nl];
}

nzl::nzl(int an)
/* initializeaza elementele lui tnz cu zilele din lunile calendaristice */
{
 int b = an%4==0&&an%100||an%400==0;

 for(int i=0;i<13;i++)
     tnz[i]=nzl::tnrz[i];
 tnz[2] += b;
```

```
}

char *nzl::retdenl(int n)
/* returneaza denumirea lunii a n-a sau pointerul nul daca n
   este in afara intervalului [1,12] */
{
 return n < 1||n > 12 ? 0: nzl::tdenluna[n];
}
```

Observaţie:

Constructorul nu testează apartenenţa anului la intervalul [1600,4900]. Acest lucru se presupune că este îndeplinit.

23.17 Să se scrie un program care citeşte un întreg de 4 cifre ce reprezintă un an calendaristic şi denumiri ale lunilor calendaristice. Pentru fiecare denumire citită, programul afişează numărul zilelor din luna respectivă.

PROGRAMUL BXXIII17

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "BXXIII16.CPP"
#include "BVIII2.CPP"
#include "BVIII3.CPP"

main()
/* - citeste un intreg de 4 cifre ce reprezinta un an calendaristic;
   - citeste denumiri ale lunilor calendaristice si afiseaza numarul zilelor
     din luna respectiva;
   - programul isi intrerupe executia la intilnirea sfirsitului de fisier. */
{
 int an;
 char d[255];

 if(pcit_int_lim("anul_calendaristic=",
      1600,4900,&an)==0){
    printf("s-a citit EOF\n");
    exit(1);
 }
 nzl nzluna(an);

 for(;;){
    printf("denumirea lunii pe un rind=");
```

```
    if(scanf("%s",d)!=1)
        break;
    int i = nzluna[d]; //determina numarul de zile din luna
    if(i)
        printf("luna:%s are %d zile\n",d,i);
    else
        printf("denumire eronata\n");
  }
}
```

23.18 Să se scrie un program care citeşte următoarele date de la intrarea standard:

- un întreg de cel mult 4 cifre care reprezintă un an calendaristic;
- 12 numere întregi care reprezintă numărul zilelor lucrătoare din lunile anului calendaristic citit anterior.

Programul afişează, pentru fiecare lună calendaristică, următoarele date:

- denumirea lunii calendaristice;
- numărul zilelor lucrătoare;
- numărul orelor efectuate de un angajat dacă se consideră că ziua lucrătoare este de 8 ore.

În final, se afişează totalul orelor efectuate de un angajat în decursul anului definit de datele citite de la intrarea standard.

În acest program funcţiile *pcit_int* şi *pcit_int_lim* sînt înlocuite de similarele lor *rcit_int* şi *rcit_int_lim*.

PROGRAMUL BXXIII18

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include "BXX4.CPP"    // rcit_int
#include "BXX5.CPP"    // rcit_int_lim
#include "BXXIII16.CPP"

main()
/* - citeste un intreg de 4 cifre care reprezinta un an calendaristic;
   - citeste 12 numere care reprezinta numarul zilelor lucratoare din
   lunile anului calendaristic citit anterior;
   - afiseaza anul calendaristic;
```

- afiseaza pentru fiecare luna:
- denumirea lunii calendaristice;
- numarul de zile lucratoare;
- numarul orelor daca se considera ca o zi lucratoare are 8 ore;
- totalul orelor lucrate din anul calendaristic.

```
*/
{
 int zile_lucrat[13];
 int total_zile = 0;
 int zi;
 int an;

 // citeste intregul care reprezinta anul calendaristic
 if(rcit_int_lim("anul calendaristic=",
      1600,4900,an)==0){
    printf("s-a tastat EOF\n");
    exit(1);
 }

 nzl nrzileimplicit(an);

 /* citeste zilele lucratoare existente in cele 12 luni
    ale anului calendaristic */
 printf("tastati zilele lucratoare\n");
 for(int i=1;i<=12;i++){
    do{
        if(rcit_int_lim(nzl::retdenl(i),0,31,zi)==
               0){
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(nrzileimplicit[i] >= zi)
            break;
        printf("numarul zilelor lucratoare\
              =%d\n",zi);
        printf("depaseste numarul zilelor din\
                luna=%d\n",
        nrzileimplicit[i]);
    }while(1);
    zile_lucrat[i]=zi;
 } // sfirsit for pentru citirea zilelor lucratoare
```

```
   // afiseaza anul calendaristic
   printf("\n anul calendaristic:%d\n",an);

   for(i=1;i<13;i++){
       // afiseaza denumirea lunii calendaristice
       printf("\n%s\n",nzl::retdenl(i));

       // afiseaza numarul de zile lucratoare
       printf("zile lucratoare=%d\n",zile_lucrat[i]);

       // afiseaza orele lucrate
       printf("ore lucrate=%d\n",zile_lucrat[i]*8);

       // insumeaza numarul zilelor lucratoare
       total_zile += zile_lucrat[i];

       printf("\n actionati o tasta pentru a\
             continua\n");
       getch();
    } // sfirsit for afisare

   // afiseaza totalul orelor lucrate
   printf("\n\n Total ore lucrate=%d\n",
           total_zile*8);
   }
```

23.19 Să se definească tipul abstract *cuvdif* pentru determinarea frecvenţelor de apariţie ale cuvintelor citite dintr-un text.

Printr-un cuvînt se înţelege o succesiune de litere. Literele mici nu se consideră distincte de cele mari.

Se consideră tipul utilizator *fcuv* definit astfel:

```
struct fcuv {
 char *cuv;
 int f;
 };
```

Pointerul *cuv* pointează spre un şir de caractere care reprezintă un cuvînt citit din textul de intrare.

Componenta *f*, numără apariţiile cuvîntului spre care pointează componenta *cuv*.

Tipul *cuvdif* se poate defini ca un tablou de elemente de tip *fcuv*. În acest caz, urmează să se indice numărul maxim de elemente al tabloului. Acesta trebuie să depăşească numărul cuvintelor diferite din textul de intrare. În cazul în care acest maxim nu este bine estimat şi el este depăşit la citirea textului de intrare, programul trebuie să se întrerupă. Pentru a evita astfel de situaţii se poate proceda ca mai jos.

În loc de a defini un tablou de tipul *fcuv* care să aibă un număr maxim de elemente, considerăm un pointer spre o zonă în care se păstrează date de tip *fcuv*. Această zonă se alocă în memoria *heap*, prin constructor, aşa încît ea să fie capabilă să păstreze un număr oarecare, *max*, de elemente.

Elementele de tip *fcuv* se păstrează în zona respectivă pe măsură ce se citesc cuvinte distincte din text. Dacă se ajunge ca zona de memorie rezervată în memoria *heap* prin constructor să fie insuficientă, acest maxim se măreşte cu o anumită cantitate, să zicem *delta*:

```
max = max+delta
```

Apoi, se alocă în memoria *heap* o zonă de memorie de dimensiune egală cu valoarea nouă a lui *max*, se copiază elementele din zona veche în cea nouă, iar zona veche se eliberează.

Componentele date ale tipului *cuvdif* sînt:

pfcuv - Este pointer spre tipul *fcuv*.

max - Este un întreg care defineşte dimensiunea zonei alocate în memoria *heap*.

ndif - Este un întreg care defineşte numărul datelor de tip *fcuv* păstrate în zona *heap*.

delta - Este un întreg care defineşte valoarea pentru incrementarea lui *max* cînd ndif > max.

Constructorul tipului abstract *cuvdif* are ca parametri valorile lui *max* şi *delta*.

Operatorul [] este supraîncărcat pentru a legaliza o expresie de forma:

```
cuvdif[cuvint]
```

unde:

cuvint - Este un pointer spre un şir de caractere care defineşte un cuvînt.

Această construcţie defineşte o referinţă la componenta *f* a elementului de tip *fcuv* pentru care *cuv* pointează spre cuvîntul identic cu cel spre care pointează *cuvint*.

Funcţia de supraîncărcare a operatorului [] realizează următoarele:

1. Caută în memoria heap un element de tip *fcuv* pentru care *cuv* pointează spre un şir de caractere identic cu cel spre care pointează *cuvint*.

   Dacă nu există un astfel de element, se continuă cu punctul 2.

   Altfel se revine din funcţie returnîndu-se o referinţă la componenta *f* a elementului respectiv.

2. Se păstrează în memoria heap un element corespunzător cuvîntului spre care pointează *cuvint*.

   f=0 pentru acest element, iar variabila *ndif* se incrementează.

   Se revine din funcţie returnîndu-se o referinţă la componenta *f* a acestui element nou păstrat în memoria *heap*.

Din cele spuse mai sus rezultă că păstrarea unui element nou în memoria *heap* este posibilă numai dacă ndif < max. Dacă această condiţie nu este îndeplinită, atunci se extinde zona cu cantitatea *delta*, aşa cum s-a indicat mai sus.

FIŞIERUL BXXIII19.H

```
struct fcuv {
 char *cuv;
 int f;
};

class cuvdif {
    fcuv *pfcuv;
    int max;
    int ndif;
    int delta;
 public:
    cuvdif(int m=100,int d=50);
    int& operator[] (const char *cuvint);
    /* returneaza o referinta la componenta f a elementului de tip fcuv
      pentru care cuv pointeaza spre acelasi sir de caractere ca si cu-
      vint */

    void afis_frecv_cuv();
    /* afiseaza frecventa cuvintelor citite */
};
```

Funcţiile membru sînt definite în fişierul de mai jos, de extensie CPP.

FIŞIERUL BXXIII19

```
#ifndef __CUVDIF_H
#include "BXXIII19.H"
#define __CUVDIF_H
#endif

cuvdif::cuvdif(int m,int d)
/* - se rezerva in memoria heap zona pentru m elemente de tip fcuv;
   - d este pasul pentru incrementarea lui max */
{
 max=m;    // maxim initial pentru cuvintele diferite
 pfcuv=new fcuv[max];
 ndif=0;   // nu sint cuvinte citite
 delta=d;  // increment pentru maxim
}

int& cuvdif::operator[] (const char *cuvint)
/* returneaza o referinta la componenta f a elementului din memoria
   heap pentru care cuv pointeaza spre un sir de caractere identic cu
   cel spre care pointeaza cuvint */
{
 /* se cauta in memoria heap un element pentru care cuv pointeaza
    spre un sir de caractere identic cu cel spre care pointeaza cuvint */

 fcuv *p;
 int i;

 for(p=pfcuv,i=0;i < ndif;i++,p++)
     if(stricmp(p -> cuv,cuvint)==0)
         return p -> f;   //s-a gasit elementul cautat

 /* nu exista in memoria heap un element pentru care cuv sa pointeze
    spre un sir de caractere identic cu cel spre care pointeaza cuvint */

 if(max==ndif){
 //se mareste zona din memoria heap
     max += delta;

     /* se rezerva zona in memoria heap pentru max elemente de tip
        fcuv */
     p = new fcuv[max];
```

```
        // se copiaza datele din zona veche in cea noua
        for(i=0;i < ndif;i++)
            p[i] = pfcuv[i];

        // se elibereaza zona veche din memoria heap
        delete pfcuv;

        // pfcuv pointeaza in continuare spre zona noua
        pfcuv = p;

    } // sfirsit if pentru incrementarea zonei din memoria heap cu
      // valoarea delta

    // rezerva zona heap pentru cuvintul spre care pointeaza cuvint
    char *c = new char[strlen(cuvint)+1];

    // transfera sirul curent in zona heap
    strcpy(c,cuvint);

    // construieste elementul din memoria heap
    p = &pfcuv[ndif++]; // adresa elementului curent din
                        // memoria heap

    // pastreaza adresa cuvintului curent
    p -> cuv = c;

    // pentru cuvintul curent f=0
    p -> f = 0;

    // returneaza o referinta la f
    return p -> f;
  }

  void cuvdif::afis_frecv_cuv()
  /* afiseaza frecventa de aparitie a cuvintelor diferite citite din textul de la
     intrarea standard */
  {
   for(int i=0;i < ndif;i++){
      printf("cuvintul:%-40s are frecventa=%d\n",
             pfcuv[i].cuv, pfcuv[i].f);
      if((i+1)%23==0){
          printf("actionati o tasta pentru a\
                 continua\n");
```

```
        getch();
      }
    }
  }
```

Observaţii:

1. În acest exemplu nu s-a testat posibilitatea depăşirii memoriei *heap*. Într-un program real este necesar ca o alocare de forma:

   ```
   p = new ...
   ```

   să se realizeze folosind un test de forma:

   ```
   if((p = new ... ) == 0){
      /* mesaj de eroare pentru depasirea memoriei heap urmat,
         eventual, de apelul functiei exit */
   }
   ```

2. În principiu, tipul abstract *cuvdif* este necesar să aibă destructor pentru a elibera zonele din memoria *heap*.

   În exemplul de faţă, se are în vedere o utilizare simplă a tipului respectiv şi nu a mai fost nevoie de destructor.

3. Atribuirea:

   ```
   p[i]=pfcuv[i]
   ```

   se realizează prin copierea implicită a datelor de tip *fcuv* (copiere bit cu bit).

4. Tipul abstract *cuvdif* este un exemplu de implementare a tablourilor dinamice.

   Un obiect care este o instanţiere a clasei *cuvdif* reprezintă un tablou dinamic alocat în memoria *heap* şi care are următoarele atribute:

   pfcuv - Este pointer spre începutul zonei alocate tabloului în care se păstrează elemente de tip *fcuv*.

   max - Este un întreg ce reprezintă dimensiunea zonei alocate în memoria *heap*.
   - Aceasta conţine *max* elemente de tip *fcuv*.

   ndif - Este un întreg ce reprezintă numărul elementelor existente în zona *heap*.

   delta - Este un întreg ce reprezintă un increment care se aplică la *max*, ori de cîte ori se depăşeşte zona din memoria *heap*.

23.20 Să se scrie un program care citeşte de la intrarea standard un text şi afişează frecvenţa de apariţie a cuvintelor diferite din textul respectiv.

Această problemă a fost rezolvată în mai multe moduri folosind:

- un tablou de pointeri;
- o listă simplu înlănţuită;
- o listă dublu înlănţuită;
- un arbore binar;
- o tabelă de dispersie.

În programul de faţă se foloseşte tipul abstract *cuvdif*. Acesta are o performanţă comparabilă cu programul implementat folosind un tablou de pointeri. Ca avantaj al programului de faţă este faptul că deşi este necesar să indicăm o dimensiune maximă pentru numărul cuvintelor distincte din textul care se citeşte, această dimensiune se ajustează automat în caz de depăşire.

PROGRAMUL BXXIII20

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include "BXXIII19.CPP"

main()
/* citeste un text si afiseaza frecventa de aparitie a cuvintelor diferite din
   textul respectiv */
{
 char t[255];
 int i;
 int c;
 cuvdif text;

 // citirea cuvintelor din text
 // textul se termina la intilnirea sfirsitului de fisier

 for(;;){
     // salt peste caractere care nu sint litere
     while((c = getchar()) != EOF && (c < 'A' ||
            c > 'Z' && c < 'a' || c > 'z'))
     ;
     if(c==EOF)
```

```
            break;
        // citeste literele unui cuvint
        for(i=0;c >= 'A' && c <= 'Z' || c >= 'a' &&
             c <= 'z';i++){
            t[i]=c;   // pastreaza o litera
            c = getchar();
        }

        // pastreaza caracterul NUL la sfirsitul cuvintului citit
        t[i]='\0';

        // incrementeaza frecventa cuvintului citit
        text[t]++;
        if(c==EOF)
            break;
    } // sfirsit for pentru citirea cuvintelor din textul de la
      // intrarea standard

    // afiseaza frecventa cuvintelor
    printf("\n Frecventa cuvintelor citite\n");
    text.afis_frecv_cuv();
}
```

Observaţie:

Expresia *text[t]* realizează apelul la funcţia pentru supraîncărcarea operatorului []. Această funcţie caută cuvîntul spre care pointează *t* în zona *heap*,corespunzătoare obiectului *text*. Dacă nu există un astfel de cuvînt, atunci acesta se păstrează în zona respectivă. Dacă nu există loc în zona alocată pentru obiectul *text*, atunci se măreşte zona respectivă. La revenirea din funcţie, se returnează o referinţă la componenta *f* corespunzătoare cuvîntului citit.

Aplicarea operatorului de incrementare expresiei *text[t]*, înseamnă incrementarea componentei *f* pentru care această expresie este o referinţă.

23.21 Să se completeze tipul abstract *cuvdif*, definit în exerciţiul 23.19., cu funcţia membru *ordcresc* care ordonează datele de tip *fcuv* din zona *heap*, aşa încît pointerii *cuv* respectivi să pointeze spre cuvintele citite în ordinea crescătoare a acestora (ordinea alfabetică).

Funcţia *ordcresc* utilizează metoda de sortare a bulelor. Pentru a obţine o sortare eficientă se vor utiliza alte metode, ca de exemplu sortarea Shell

sau rapidă (vezi capitolul 14).

FIŞIERUL BXXIII21.H

```
struct fcuv {
 char *cuv;
 int f;
};
class cuvdif {
     fcuv *pfcuv;
     int max;
     int ndif;
     int delta;
 public:
     cuvdif(int m=100,int d=50);
     int& operator[](const char *cuvint);
     void ordcresc();
     /* ordoneaza elementele din zona heap in asa fel incit cuvintele
        spre care pointeaza cuv sa formeze un sir de cuvinte ordonat
        crescator (in ordine alfabetica) */

     void afis_frecv_cuv();
};
```

Funcţiile membru, exceptînd *ordcresc*, sînt definite în fişierul BXXIII19.CPP.

În fişierul de mai jos se adaugă definiţia funcţiei *ordcresc* la definiţiile existente în fişierul BXXIII19.CPP.

Pentru detalii în legătură cu metoda de sortare a bulelor se poate consulta capitolul 14.

FIŞIERUL BXXIII21

```
#ifndef __CUVDIF_H
#include "BXXIII21.H"
#define __CUVDIF_H
#endif
#include "BXXIII19.CPP"

void cuvdif::ordcresc()
/* ordoneaza elementele din memoria heap in asa fel incit cuvintele spre
   care pointeaza cuv sa formeze un sir de cuvinte ordonat crescator */
{
 int i,ind;
```

```
    ind = 1;
    while(ind){
        ind = 0;
        for(i=0;i < ndif-1;i++)
            if(stricmp(pfcuv[i].cuv,
                         pfcuv[i+1].cuv) > 0){
                fcuv temp;
                temp = pfcuv[i];
                pfcuv[i]=pfcuv[i+1];
                pfcuv[i+1]=temp;
                ind = 1;
            }
        }
}
```

23.22 Să se scrie un program care citeşte cuvintele dintr-un text şi le afişează în ordine alfabetică împreună cu frecvenţa lor de apariţie.

Programul de faţă este similar cu cel din exerciţiul 23.20.

PROGRAMUL BXXIII22

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include "BXXIII21.CPP"

main()
/* citeste cuvintele din textul de la intrarea standard si le afiseaza in
   ordine alfabetica impreuna cu frecventa lor de aparitie */
{
 char t[255];
 int i;
 int c;
 cuvdif text;

 for(;;) {
    while((c=getchar())!=EOF&&(c < 'A'||c > 'Z' &&
             c < 'a' || c > 'z'))
      ;
    if(c == EOF)
       break;
    for(i=0;c >= 'A' && c <= 'Z' ||
            c >= 'a' && c <= 'z';i++){
```

```
            t[i]=c;
            c=getchar();
        }
        t[i]='\0';
        text[t]++;
        if(c == EOF)
            break;
    }

    //sortare in ordine alfabetica
    text.ordcresc();
    text.afis_frecv_cuv();
}
```

23.4. Supraîncărcarea operatorului () (Operatorul de apel funcţie)

O funcţie se apelează printr-o construcţie de forma:

(1) *nume_functie(lista_parametrilor_efectivi)*

Această construcţie poate fi privită ca un tot unitar care să formeze un operand. În realitate, ea este o expresie formată din doi operanzi:

nume_functie

şi

lista_parametrilor_efectivi

la care se aplică operatorul binar ().

Ceea ce este specific pentru această interpretare este faptul că cel de al doilea parametru (*lista_parametrilor_efectivi*) poate fi şi vid. Aceasta înseamnă că operatorul () poate avea ca operand secund un operand vid, lucru care nu este posibil pentru ceilalţi operatori binari.

Apelul unei funcţii se poate realiza folosind formatul de mai jos, în care numele funcţiei se înlocuieşte cu o expresie de tip pointer spre funcţia respectivă:

(2) (*ep)(*lista_parametrilor_efectivi*)

Acest apel se utilizează în cazul în care nu se cunoaşte numele funcţiei, ci numai un pointer spre funcţia respectivă (vezi capitolul 8).

Operatorul () se poate supraîncărca aşa încît primul operand să fie un obiect:

(3) *obiect(lista_parametrilor_efectivi)*

O astfel de expresie este echivalentă cu un apel de forma:

obiect.operator() (lista_parametrilor_efectivi)

Funcţia care supraîncarcă operatorul () trebuie să fie o funcţie membru nestatică.

Lista_parametrilor_efectivi se tratează ca în apelurile obişnuite de funcţii.

Supraîncărcarea operatorului () se utilizează frecvent la definirea aşa numitului *iterator*.

Iteratorii se utilizează în legătură cu tipuri abstracte care conţin colecţii de elemente. Astfel de tipuri de date sînt listele, arborii, tabelele de dispersie etc.

Problema consultării (căutării) elementelor dintr-o colecţie de elemente protejate se rezolvă cel mai simplu cu ajutorul iteratorilor. În afară de faptul că iteratorii asigură protecţia datelor, ei oferă un mijloc simplu de acces la elementele unei colecţii fără a intra în detaliile legate de implementarea colecţiei, mod de ordonare a elementelor colecţiei etc. Aceasta asigură o independenţă mare a utilizării colecţiei de implementarea ei.

În principiu, un iterator se implementează printr-o clasă specială ataşată unui tip abstract care conţine o colecţie de elemente. Fie, de exemplu, o clasă pe care o numim *container* şi care este o colecţie de obiecte de un tip oarecare.

Colecţia de obiecte poate fi implementată în mai multe moduri:

- tablou de obiecte de tip *obiect*, de dimensiune fixă sau variabilă;
- listă de noduri de tip *obiect* simplu sau dublu înlănţuită;
- arbore binar cu noduri de tip *obiect*, tabelă de dispersie cu înregistrări de tip *obiect* etc.

În funcţie de organizarea aleasă pentru implementarea tipului *container*, clasa *container* are o dată membru protejată de tip *obiect* sau pointer spre *obiect*.

De exemplu, dacă tipul *container* se implementează cu un tablou de dimensiune fixă, atunci clasa *container* se defineşte astfel:

(a)
```
class container {
   obiect tab[n];
   ...
};
```

unde:

n - Este o constantă întreagă.

În acest caz, un iterator va asigura acces la elementele tab[0],tab[1],... tab[n-1]. Accesul se realizează secvenţial:la primul apel se permite acces la tab[0], apoi la tab[1] şi aşa mai departe.

În cazul în care tipul *container* se implementează ca o colecţie dinamică de elemente de tip *obiect*, se poate proceda ca în cazul tipului abstract *cuvdif*, implementat în exerciţiul 23.19. Clasa *container* se defineşte astfel:

```
(b) class container {
     obiect *colectie;
     int max;
     int crt;
     int delta;
     ...
    };
```

În acest caz, elementele de tip *obiect* se păstrează într-o zonă din memoria *heap*. Dimensiunea curentă a acestei zone este stabilită în aşa fel încît ea să poată conţine pînă la *max* obiecte de tip *obiect*.

Pointerul *colectie* pointează spre începutul acestei zone. Variabila *crt* are ca valoare numărul obiectelor de tip *obiect* prezente în memoria *heap*.

În cazul în care zona rezervată în memoria *heap* devine insuficientă, se rezervă o altă zonă de dimensiune *max+delta* în care se transferă obiectele existente, aşa cum s-a procedat în exerciţiul 23.19.

Ca şi în cazul (a), iteratorul va asigura acces succesiv la obiectele: colectie[0], colectie[1],...,colectie[crt-1].

Dacă tipul *container* se implementează ca o listă simplu înlănţuită, atunci nodurile se definesc ca în capitolul 11:

struct *obiect* {
  declaratii
  struct *obiect* **urm*;
};

Clasa *container* are ca şi date membru pointerii *prim* şi *ultim*, care pointează spre capetele listei. Aceşti pointeri au fost definiţi în capitolul 11 ca variabile globale pentru implementarea listelor simplu şi dublu înlănţuite. Clasa *container* se poate defini astfel:

```
(c) class container {
      obiect *prim,*ultim;
      ...
    };
```

În acest caz, un iterator trebuie să asigure acces succesiv la nodurile listei, începînd cu cel spre care pointează *prim* şi terminînd cu cel spre care pointează *ultim*.

În cazul listelor dublu înlănţuite se pot defini doi iteratori, pentru a permite accese secvenţiale la nodurile listei în ambele sensuri.

Dacă tipul *container* se implementează ca un arbore, atunci nodurile se pot defini ca în capitolul 12:

struct *obiect* {
    declaratii
    struct *obiect* ***st**;
    struct *obiect* ***dr**;
};

Clasa *container* are, în acest caz, ca dată membru un pointer spre rădăcina arborelui, numit *prad*:

```
(d) class container {
      obiect *prad;
      ...
    };
```

În acest caz, se pot defini 3 tipuri de iteratori care să permită accesul la obiectele colecţiei în *preordine*, *inordine* şi *postordine*.

În cazul în care tipul *container* se implementează printr-o tabelă de dispersie, se poate utiliza clasa:

```
(e) class container {
      obiect *thash[M];
      ...
    };
```

unde:

M - Este o constantă întreagă şi care, de obicei, este un număr prim.

Tipul *obiect* se defineşte la fel ca nodurile listelor simplu înlănţuite deoarece *thash* este un tablou de pointeri spre nodurile de început ale listelor simplu înlănţuite care sînt alcătuite din noduri ale căror chei intră în coliziune (vezi cap.13).

În acest caz, clasa *container* are o funcţie membru pentru calculul *dispersiei*. Iteratorul pentru clasa *container* poate fi implementat în mai multe moduri. Astfel, se poate defini un iterator pentru a permite acces la nodurile unei liste spre începutul căreia pointează un element al tabloului *thash*. Un astfel de iterator poate avea ca parametru un întreg din intervalul [0,M-1] care este un indice în tabloul *thash* şi defineşte elementul pointer spre începutul listei ale cărei noduri sînt accesate prin intermediul iteratorului.

O altă posibilitate este ca iteratorul să aibă ca parametru un obiect, iar acesta realizează acces succesiv la obiectele care se află în coliziune cu parametrul. În acest caz, iteratorul apelează funcţia membru a clasei container pentru calculul dispersiei şi determină, în felul acesta, indicele din tabela *thash* care defineşte pointerul spre lista cu obiectele aflate în coliziune şi care sînt accesate prin iterator.

În sfîrşit, să amintim că se poate defini un iterator care să realizeze accesul succesiv la obiectele din listele spre care pointează elementele tabloului *thash*, începînd cu pointerul thash[0], continuînd cu tahsh[1], thash[2] şi aşa mai departe pînă la thash [M-1].

Din cele de mai sus, rezultă că un iterator are următoarele proprietăţi:

- Este ataşat unui tip abstract care corespunde unei colecţii de obiecte (mai sus, un astfel de tip a fost numit *container*. O instanţiere a clasei *container* este un obiect care este o colecţie de obiecte de tip *obiect*).
- Este capabil să livreze, pe rînd, obiectele elemente ale colecţiei.

Avînd în vedere caracteristicile iteratorilor, putem face o analogie dintre aceştia şi operaţia de citire a fişierelor.

Operaţia de citire a fişierelor se realizează apelînd o funcţie specială care, la fiecare apel, livrează înregistrarea următoare din fişier. Iteratorul, la fel ca şi funcţia de citire, are drept scop să asigure accesul la obiectul următor din colecţie.

Citirile înregistrărilor dintr-un fişier trebuie să fie precedate de operaţia de deschidere a fişierului respectiv. În mod analog, în cazul iteratorilor este necesară o operaţie care să definească obiectul la care se asigură accesul la primul apel al iteratorului. Această operaţie se numeşte *iniţializarea* iteratorului.

Un mod simplu de a realiza un iterator este acela de a implementa conceptul de iterator printr-un tip abstract. Un astfel de tip este strîns legat de clasa care defineşte colecţia de obiecte pentru care se doreşte să se definească accesul.

Mai sus, am numit *container* o astfel de clasă. Numim *iterator* clasa ataşată clasei *container* şi care defineşte conceptul de iterator pentru ea.

Funcţiile membru ale clasei *iterator*, de obicei, trebuie să aibă acces la elementele protejate ale clasei *container* şi de aceea, ea este o clasă prieten pentru clasa *container*. Acest lucru se defineşte simplu, de exemplu, clasa *container* de la punctul (b) de mai sus,se completează cu o declaraţie corespunzătoare, ca mai jos:

```
class iterator;   // definitie incompleta
class container {
    obiect *colectie;
    ...
    friend class iterator;
};
```

Iniţializarea iteratorului este în funcţie de modul în care se defineşte colecţia de obiecte. În toate cazurile, ea se realizează cu ajutorul constructorului clasei *iterator*.

Iteratorul clasei *container* se aplică la instanţierile clasei *container*. De aceea, o instanţiere a clasei *iterator* se realizează pentru o instanţiere a clasei *container*. De aici rezultă că, o instanţiere a clasei *iterator* trebuie să conţină un pointer sau o referinţă la obiectul curent de tip *container* (pentru care se realizează iterarea).

Aceasta înseamnă că, o clasă *iterator* are o dată membru de forma:

```
container *crtcontainer;
```

sau

```
container& crtcontainer;
```

O altă dată membru a unei clase iterator este destinată pentru a defini starea iteraţiei. Această dată defineşte obiectul curent din container. Ea este actualizată la fiecare apel al iteratorului. Această dată este dependentă de modul în care este definită colecţia la care este ataşat *iteratorul*. În exemplul de mai sus, data respectivă poate fi un întreg care defineşte indicele obiectului curent, deci ea va fi declarată ca mai jos:

```
int i;
```

Ambele date membru ale clasei iterator se vor iniţializa cu ajutorul constructorului clasei *iterator*.

Pentru a iniţializa prima dată membru, constructorul are un parametru de tip pointer sau referinţă la tipul container:

```
container *pcontainer
```

sau

```
container& rcontainer
```

A doua dată membru, de obicei, are o iniţializare implicită, dar pot apare situaţii cînd se doreşte să se impună obiectul din container de la care să se pornească iteraţia. De aici rezultă că clasa *iterator*, pentru exemplul de mai sus, este de forma:

```
class iterator {
     const container *crtcontainer;
     int i;
  public:
     iterator(const container& crtinst,
     int indice = 0)
     {
         crtcontainer = &crtinst;
         i = indice;
     }
     ...
};
```

Funcţia membru care permite accesul la obiectele colecţiei *container* este o funcţie care supraîncarcă operatorul (). La fiecare apel al ei, se returnează un pointer sau o referinţă la obiectul curent din *container*. Totodată, se actualizează data membru a clasei *iterator*, pentru ca la apelul următor, să se returneze pointerul sau referinţa la obiectul următor din *container*.

În cazul clasei *iterator* de mai sus, vom utiliza următoarea funcţie membru pentru supraîncărcarea operatorului ():

```
obiect *operator()()
{
 return i < crtcontainer -> crt ? & crtcontainer ->
            colectie[i++] : 0;
}
```

Fie instanţierile:

```
container colectie_obiecte;
```

şi

```
iterator iterator_colectie(colectie_obiecte);
```

După memorarea obiectelor în obiectul *colectie_obiecte* de tip container, conţinutul acestora se poate afişa folosind instanţierea *iterator_colectie* ataşată obiectului *colectie_obiecte*. În acest scop se apelează funcţia care

supraîncarcă operatorul () pentru obiectul *iterator_colectie*. Apelul se realizează cu ajutorul expresiei:

```
iterator_colectie()
```

care poate fi folosit ca un operand obişnuit al unei expresii. Apelul de mai sus, returnează un pointer spre o dată de tip *obiect*.

Fie, de exemplu, declaraţia:

```
obiect *p;
```

atunci un ciclu de forma:

```
while((p = iterator_colectie())!=0){
  ...
}
```

atribuie succesiv lui *p*, pointerii spre obiectele păstrate în obiectul *colectie_obiecte*.

Se observă avantajul utilizării iteratorilor pentru obiectele de tip *container*. Aceştia permit accesul la obiectele din container fără a avea în vedere detaliile legate de modul în care se realizează trecerea de la un obiect al containerului la un altul.

Propunem cititorului să implementeze clasa *iterator* şi pentru celelalte variante de implementare a clasei *container*.

În încheierea acestui paragraf vom observa că, supraîncărcarea operatorului () permite utilizatorului să folosească construcţii asemănătoare cu cele pentru apelul funcţiilor, înlocuind numele funcţiei prin obiecte.

În rest, se utilizează aceleaşi reguli pentru transferul parametrilor şi returnarea rezultatelor ca în cazul apelurilor obişnuite de funcţii.

Exerciţii:

23.23 Fie clasa *cuvdif*, definită în exerciţiul 23.21. Să se definească o clasă *iterator* pentru clasa *cuvdif*.

Numim *itercuvdif* clasa iterator ataşată clasei *cuvdif*. Clasa *itercuvdif* se defineşte după modelul clasei *iterator* ataşată clasei *container* în paragraful de faţă.

FIŞIERUL BXXIII23.H

```
struct fcuv {
  char *cuv;
  int f;
```

```
};
class itercuvdif;

class cuvdif {
     fcuv *pfcuv;
     int max;
     int ndif;
     int delta;
 public:
     cuvdif(int m=100,int d=50);
     int& operator[](const char *cuvint);
     void ordcresc();
     void afis_frecv_cuv();
     friend class itercuvdif;
};
class itercuvdif {
     const cuvdif *container;
     int i;
 public:
     itercuvdif(const cuvdif& crtcontainer,
                int indice=0);
     fcuv *operator()();
};
```

Funcţiile membru, definite în fişierul BXXIII21.CPP, rămîn valabile şi în cazul de faţă. Se mai adaugă funcţiile membru ale clasei iterator:

- constructorul iteratorului

şi

- funcţia pentru supraîncărcarea operatorului ().

FIŞIERUL BXXIII23

```
#ifndef __CUVDIF_H
#include "BXXIII23.H"
#define __CUVDIF_H
#endif
#include "BXXIII21.CPP"

inline itercuvdif::itercuvdif(const cuvdif& crt,
                              int indice)
{
 container = &crt;
```

```
 i = indice;
}

inline fcuv *itercuvdif::operator()()
{
 return i < container -> ndif ? & container ->
            pfcuv[i++]:0;
}
```

23.24 Să se scrie o funcţie care citeşte un cuvînt de la intrarea standard. Funcţia returnează valoarea 0 la întîlnirea sfîrşitului de fişier sau codul ASCII al caracterului citit care urmează imediat după cuvîntul citit.

FUNCŢIA BXXIII24

```
int citcuv(char *cuv)
/* - citeste un cuvint de la intrarea standard;
   - returneaza:
        - 0 la intilnirea sfirsitului de fisier;
        - codul ASCII al caracterului citit care urmeaza imediat
          dupa cuvintul citit
*/
{
 int i,c;
 i = 0;

 // salt peste caractere care nu sint litere
 while((c=getchar()) != EOF &&
     (c < 'A' || c > 'Z' && c < 'a' || c > 'z'))
    ;
 if(c == EOF) {          // s-a intilnit EOF
    cuv[0] = '\0';      // cuvintul vid
    return 0;
 }

 // pastreaza caracterele cuvintului curent
 while(c >= 'A' && c <= 'Z' || c >= 'a' &&
      c <= 'z'){
    cuv[i++]=c;
    c=getchar();
 }
 cuv[i]='\0';
 if(c==EOF)
    return 0;
```

```
    else
       return c;
  }
```

23.25 Să se scrie un program care citeşte cuvintele dintr-un text şi le afişează în ordine alfabetică împreună cu frecvenţa lor de apariţie.

Programul de faţă este similar cu cel din exerciţiul 23.22. Diferenţa esenţială dintre ele constă în aceea că programul de faţă foloseşte clasa *itercuvdif* pentru a lista frecvenţa cuvintelor, spre deosebire de programul amintit mai sus, care foloseşte, în acelaşi scop, funcţia membru *afis_frecv_cuv*.

PROGRAMUL BXXIII25

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include "BXXIII23.CPP"
#include "BXXIII24.CPP"

main()
/* citeste cuvintele din textul de la intrarea standard si le afiseaza in
   ordine alfabetica impreuna cu frecventa lor de aparitie
*/
{
 char t[255];
 cuvdif text;
 int i;

/* citeste cuvintele din textul de la intrarea standard si determina
   frecventa lor de aparitie
*/
 for(;;){
     i=citcuv(t);
     if(i==0&&strlen(t)==0)
        break;        // sfirsitul textului
     text[t]++;
     if(i==0)
        break; // sfirsitul textului
}

  // sorteaza cuvintele citite in ordine alfabetica
 text.ordcresc();
```

```
   // afiseaza frecventa cuvintelor citite
   itercuvdif obiter(text); /* se ataseaza un obiect de tip
                               iterator pentru obiectul text */
   fcuv *p;
   i=0;
   while((p=obiter())!=0){
      printf("cuvintul: %-40s are frecventa= %d\n",
             p -> cuv,p -> f);
      if(++i%23==0){
         printf("Actionati o tasta pentru a\
                continua\n");
         getch();
      }
   }
}
```

23.26 Tipul definit de utilizator, *fcuv*, care a fost declarat în exercițiul 23.23. se modifică astfel:

```
struct fcuv {
    char *cuv;
    int f;
    struct fcuv *urm;
};
```

unde:

cuv și *f* - Au aceleași semnificații ca în cazul exercițiului 23.23.

Să se modifice clasele *cuvdif* și *itercuvdif* ținînd seama de noua definiție a tipului *fcuv*.

Clasele *cuvdif* și *itercuvdif* au aceleași funcții *membru*. Acestea se utilizează în același mod, deci realizează aceeași interfață cu programul care le utilizează ca și cele definite în fișierele BXXIII19.CPP, BXXIII21.CPP și BXXIII23.CPP.

FIȘIERUL BXXIII26.H

```
struct fcuv {
 char *cuv;
 int f;
 struct fcuv *urm;
};

class itercuvdif;
```

```
class cuvdif {
    fcuv *prim,*ultim;
  public:
    cuvdif();
    int& operator [] ( char *cuvint);
    void ordcresc();
    void afis_frecv_cuv();
    friend class itercuvdif;
};

class itercuvdif {      // clasa iterator
    fcuv *crt;
  public:
    itercuvdif(const cuvdif& inceput);
    fcuv *operator () ();
};
```

Mai jos, se definesc funcţiile membru într-un fişier de extensie *CPP*.

Clasa *cuvdif* implementează tipul abstract *cuvdif* printr-o listă simplu înlănţuită. Pentru detalii în legătură cu listele simplu înlănţuite se poate consulta capitolul 11.

Constructorul *cuvdif* iniţializează pointerii *prim* şi *ultim* cu pointerul nul.

Funcţia pentru supraîncărcarea operatorului [] realizează o căutare liniară în listă a şirului spre care pointează *cuvint*. Dacă nu există, în listă, un nod în aşa fel încît pointerul *cuv* să pointeze spre un şir de caractere identic cu cel spre care pointează *cuvint*, atunci se adaugă un astfel de nod la listă.

Funcţia returnează o referinţă la componenta *f* a nodului căutat sau adăugat la listă.

Funcţia *ordcresc*, ordonează nodurile în listă în aşa fel încît şirurile de caractere spre care pointează componenta *cuv* să fie în ordine crescătoare.

FIŞIERUL BXXIII26

```
#ifndef __CUVDIF_H
#include "BXXIII26.h"
#define __CUVDIF_H
#endif

inline cuvdif::cuvdif()
// initializeaza obiectul cu lista vida
{
 prim=ultim=0;
```

```
}

int & cuvdif::operator [] (char *cuvint)
/* - cauta un nod in lista asa incit componenta cuv a nodului sa pointeze
     spre un sir identic cu sirul spre care pointeaza cuvint; daca exista
     un astfel de nod, atunci se returneaza o referinta la componenta f
     a nodului determinat in acest fel;
   - daca nu se gaseste un nod ca cel indicat mai sus, atunci se adauga la
     lista un nod corespunzator sirului spre care pointeaza cuvint, se
     atribuie valoarea zero componentei f a nodului adaugat si se
     returneaza o referinta la aceasta componenta.
*/
{
 fcuv *p;
 static char er[]="memorie insuficienta\n";

 // cautare liniara in lista
 for(p=prim;p;p=p -> urm)
    if(stricmp(p -> cuv,cuvint) == 0)
        return p -> f;

 /* nu s-a gasit nodul cautat in lista; se adauga un nod corespunzator
    sirului spre care pointeaza cuvint */
 char *q;

 // se pastreaza sirul in memoria heap
 if((q=new char [strlen(cuvint)+1]) == 0){
    printf(er);
    exit(1);
 }
 strcpy(q,cuvint);

 // rezerva zona pentru nod
 if((p=new fcuv)==0){
    printf(er);
    exit(1);
 }

 /* construieste si adauga nodul corespunzator sirului spre care
    pointeaza cuvint
 */
 p -> cuv=q;
 p -> f=0;
```

```
 p -> urm=0;
 if(prim == 0)    // lista vida
    prim=p;
 else
 /* se adauga nodul dupa nodul spre care pointeaza ultim */
 ultim->urm=p;
 ultim=p;     // ultim pointeaza spre nodul adaugat
 return p -> f;

} // sfirsit operator []

void cuvdif::ordcresc()
/* - ordoneaza nodurile listei in asa fel incit sirurile spre care pointeaza
     componenta cuv a nodurilor sa fie in ordine crescatoare;
   - ordonarea se realizeaza prin modificarea corespunzatoare a
    inlantuirii nodurilor;
   - in acest scop se utilizeaza metoda de sortare a bulelor.
*/
{
 fcuv *p,*q,*r;
 int ind;

 if(prim==0)
     return; // lista vida
 if(prim==ultim)
     return;   // lista are un singur nod
 ind=1;
 while(ind){
     ind=0;

     /* p,q,r pointeaza spre trei noduri consecutive ale listei */

     for(p=0,q=prim,r=q -> urm;r;)
         if(stricmp(q -> cuv,r -> cuv)>0){
             if(p==0)     /* se permuta primul nod al listei */
                 prim = r;
             else
                 p -> urm=r;
             q -> urm = r -> urm;
             r -> urm=q;
             p=r;
             r=q -> urm;
             if(r == 0)     /* s-a ajuns la ultimul nod al listei */
```

```
                ultim = q;
            ind = 1;     /* deoarece s-a facut permutare */
        } else {      // nu se face permutare
        // se avanseaza in lista
        p=q;
        q=r;
        r=r -> urm;
    } // sfirsit for
  } // sfirsit while
} // sfirsit ordcresc

void cuvdif::afis_frecv_cuv()
/* afiseaza frecventa de aparitie a cuvintelor diferite citite de la intrarea
   standard
*/
{
 fcuv *p;
 int i=0;

 for(p=prim;p;p=p -> urm){
     printf("cuvintul: %-40s are frecventa= %d\n",
             p -> cuv,p -> f);
     if(++i%23 == 0){
        printf("Actionati o tasta pentru a\
              continua\n");
        getch();
     }
 }
} // sfirsit afis_frecv_cuv

inline itercuvdif::itercuvdif(const
                                       cuvdif& inceput)
// constructor pentru iterator
{
 crt=inceput.prim;
}

fcuv *itercuvdif::operator () ()
/* - supraincarca operatorul (); permite acces la nodul curent din lista;
   - returneaza pointerul la nodul curent din lista sau 0 daca nu exista un
     astfel de nod.
*/
{
```

```
fcuv *crtl=crt; /* pastreaza pointerul spre nodul curent */
if(crt) // avans la nodul urmator
    crt=crt -> urm;
return crtl;
}
```

Observaţii:

1. Funcţia membru *afis_frecv_cuv* este dependentă de structura nodului care se defineşte ca o dată de tip *fcuv* şi care este un tip utilizator.

   Dacă *fcuv* se schimbă, atunci funcţia *afis_frecv_cuv* poate să nu mai fie necesară sau trebuie schimbată. De aceea, de obicei se renunţă la o funcţie membru de acest fel şi acţiunea realizată de ea se poate face folosind iteratorul. Aşa cum s-a văzut mai sus, clasa iterator supraîncarcă operatorul () şi funcţia membru respectivă permite acces secvenţial la datele de tip *fcuv*. Acestea sînt neprotejate şi deci se poate face acces la componentele lor în mod obişnuit.

2. În mod obişnuit, o clasă iterator are două componente "dată membru": una care este o referinţă la obiectul colecţie curent pentru care se aplică iteratorul şi alta care defineşte obiectul curent din container.

   În cazul de faţă este suficientă o singură dată membru şi anume pointerul spre nodul curent. Aceasta, deoarece pointerul respectiv, permite el singur accesul la nodul următor din listă.

3. Programul din exerciţiul 23.25., se poate modifica pentru a utiliza tipurile abstracte *cuvdif* şi *itercuvdif* definite mai sus. În acest scop, este necesar să se includă fişierul BXXIII26.CPP în locul fişierului BXXIII23.CPP. Alte modificări nu sînt necesare deoarece funcţiile membru ale tipurilor abstracte de mai sus, au aceeaşi utilizare indiferent de faptul că sînt incluse dintr-un fişier sau altul. Ele realizează aceeaşi interfaţă pentru programul din exerciţiul 23.25.

Propunem cititorului să realizeze aceleaşi tipuri abstracte folosind alte implementări, cum ar fi:

- listă dublu înlănţuită;
- arbore binar;
- tabelă de dispersie.

23.5. Supraîncărcarea operatorului ->

Supraîncărcarea operatorului -> se realizează printr-o funcţie membru

nestatică. La supraîncărcare, acest operator este considerat ca fiind un operator unar care se aplică la operandul care îl precede.

Fie expresia:

(1) `obiect -> expresie`

unde:

obiect - Este o instanţiere a clasei *cl*.

Această expresie se evaluează înlocuind operandul *obiect* cu apelul funcţiei membru care supraîncarcă operatorul -> pentru obiectele clasei *cl*. Cu alte cuvinte, expresia de mai sus devine:

(2) `(obiect.operator -> ()) -> expresie`

În continuare, evaluarea expresiei este dependentă de tipul de dată returnat de apelul:

(3) `obiect.operator -> ()`

Dacă acest apel returnează un pointer spre o dată oarecare, atunci se aplică operatorul predefinit ->, adică se selectează componenta definită de *expresie* a datei spre care pointează pointerul (3).

Dacă apelul (3) returnează un obiect, să zicem *obiect1*, care este o instanţiere a clasei *cl1*, atunci expresia (2) devine de forma (1) de mai sus:

```
obiect1 -> expresie
```

şi în continuare urmează a se face evaluarea apelîndu-se pentru *obiect1*, funcţia membru a clasei *cl1* care supraîncarcă operatorul ->. Dacă nu există o astfel de supraîncărcare, atunci expresia iniţială este eronată.

Exemplu:

Fie tipul utilizator:

```
struct fcuv {
 char *cuv;
 int f;
 struct fcuv *urm;
};
```

Definim un tablou de pointeri spre elementele de tip *fcuv*:

```
fcuv *tab[MAX];
```

Expresia:

```
tab[i] -> cuv
```

utilizează operatorul -> predefinit. Ea defineşte şirul de caractere spre care pointează *cuv*, componentă a datei structurate de tip *fcuv* spre care pointează tab [i].

Fie clasa:

```
class nod {
    fcuv *pe;
    ...
 public:
    nod(char *p);
    fcuv *operator -> ()
    {
        return pe;
    }
    ...
};
```

Constructorul alocă în memoria *heap* o dată de tip *fcuv* şi *pe* pointează spre ea:

```
nod::nod(char *p)
{
 pe = new fcuv;

 // mai jos se aplica operatorul -> predefinit
 pe -> cuv = new char [strlen(p)+1];
 strcpy(pe -> cuv,p);
 pe -> f = 1;
 pe -> urm = 0;
}
```

Fie instanţierea:

```
nod crt("apa");
```

Ea aplică constructorul *nod* şi se păstrează în memoria *heap* o dată de tip *fcuv*.

Pointerul *pe* pointează spre această dată. El este componentă a instanţierii *crt*, deci *crt.pe* pointează spre data de tip *fcuv* păstrată în memoria *heap*. Această dată are componentele iniţializate de constructor astfel:

cuv	- Pointează spre şirul de caractere "apa", păstrat şi el în memoria heap,
f	- Are valoarea 1.

urm - Are valoarea zero (pointerul nul).

În concluzie, expresia:

crt.pe -> cuv

pointează spre şirul "apa" păstrat în memoria *heap*. Cum *pe* este o dată membru protejată, expresia de mai sus poate fi utilizată numai în corpul unei funcţii membru sau prieten al clasei *nod*.

Expresia:

(4) `crt -> cuv`

este legală şi ea se interpretează ca mai jos:

operatorul -> se aplică instanţierii *crt* a clasei *nod*.

Deoarece pentru clasa *nod* există o supraîncărcare a operatorului ->, expresia de mai sus devine:

```
(crt.operator -> ()) -> cuv
```

Apelul:

```
crt.operator -> ()
```

returnează valoarea pointerului *pe*, deci expresia (4) este echivalentă cu expresia:

```
crt.pe -> cuv
```

Aceasta, am văzut că pointează spre şirul de caractere "apa", păstrat prin intermediul constructorului în memoria *heap*.

Expresia:

```
crt -> cuv
```

nu este protejată şi prin urmare ea poate fi utilizată în mod obişnuit. De exemplu, instrucţiunea:

```
printf("%s",crt -> cuv);
```

este legală şi va afişa cuvîntul:

apa

în poziţia curentă a cursorului pe ecran.

Din acest exemplu se vede că expresia crt.pe ->cuv are aceeaşi valoare ca şi expresia crt -> cuv în urma supraîncărcării operatorului -> pentru clasa *nod*.

Diferenţa dintre ele se referă numai la modul de utilizare. Astfel, expresia crt.pe ->cuv se poate utiliza numai în corpul unei funcţii membru sau prieten al clasei *nod*, în timp ce expresia crt -> cuv nu este protejată. Acelaşi lucru este valabil şi pentru expresiile crt.pe ->f şi crt -> f.

24. CONVERSII

Compilatoarele C şi C++ execută în mod automat o serie de conversii pentru datele de tipuri predefinite. Aceste conversii intervin în următoarele situaţii:

1. Aplicarea unui operator la operanzi de tipuri diferite.
2. Parametrul efectiv al unei funcţii are un tip diferit de parametrul formal care îi corespunde.
3. Tipul din antetul unei funcţii diferă de cel al expresiei aflate în instrucţiunea *return* care realizează revenirea din funcţie.

Conversiile de la punctul 1, se execută folosind regula conversiilor implicite dacă operatorul nu este de atribuire.

În principiu, această regulă se aplică în aşa fel încît operandul de tip "inferior" să se convertească spre tipul "superior" al celuilalt operand. Pentru detalii se poate consulta paragraful 3.2.2.

În cazul operatorului de atribuire, valoarea expresiei din dreapta semnului de atribuire se converteşte spre tipul expresiei din stînga semnului de atribuire.

În cazul conversiilor de la punctul 2, valoarea parametrului efectiv se converteşte spre tipul parametrului formal care îi corespunde.

Conversiile de la punctul 3, se aplică pentru a converti tipul expresiei din instrucţiunea *return* spre tipul care se află în antetul funcţiei.

Conversiile pot conduce la trunchieri, exceptînd cele care se realizează la punctul 1 (regula conversiilor implicite).

Conversia unei valori de tip "inferior" spre un tip "superior" se face fără trunchiere.

Amintim că utilizatorul poate, în anumite cazuri, să impună realizarea unor conversii cu ajutorul operatorului de forţare a tipului sau de conversie explicită (vezi paragraful 3.2.9.). Acest operator se utilizează în limbajul C în expresii de forma:

(tip)operand

Expresiile de această formă se numesc expresii *cast*, iar operatorul:

(tip)

îl vom numi şi operator *cast*.

Conversia explicită este necesară adesea pentru evaluarea corectă a expresiilor.

De exemplu, dacă *n* este o variabilă de tip *int* şi se doreşte valoarea raportului dintre *n* şi *n+1*, atunci nu putem utiliza expresia:

n/(n+1)

deoarece ambii operanzi fiind de tip *int*, se realizează împărţirea întreagă şi rezultatul va fi egal cu zero. Valoarea raportului respectiv se obţine dacă, de exemplu, se converteşte primul operand spre tipul *double* folosind expresia *cast*:

(double)n

Expresia devine:

(double)n/(n+1)

care se evaluează astfel:

- întîi se converteşte *n* spre tipul *double*, deoarece operatorul *cast* este prioritar;
- apoi se evaluează suma n+1;
- urmează să se aplice operatorul de împărţire pentru doi operanzi de tipuri diferite:
  - deîmpărţitul este de tip *double*,
  - împărţitorul de tip *int*;

  conform regulei conversiilor implicite, împărţitorul se converteşte spre tipul *double* şi apoi se realizează împărţirea flotantă şi se obţine rezultatul dorit.

Există şi alte situaţii în care este obligatorie utilizarea operatorului *cast*. Aşa de exemplu, la o atribuire a unui pointer spre *void* la un alt pointer, care este spre un tip diferit de *void*.

Exemplu:

```
int *p;
void *v;
...
p = (int *)v;
```

În limbajul C++ expresiile *cast* au şi un alt format decît cel indicat mai sus şi anume:

tip(*expresie*)

Efectul acestei expresii constă în aceea că, valoarea expresiei din

paranteză se converteşte spre tipul *tip*.

Utilizarea obiectelor conduc şi ele la situaţii de felul celor de mai sus în care sînt necesare conversii.

Datorită posibilităţii supraîncărcării operatorilor, pot apare situaţii în care un operator trebuie să se execute asupra operanzilor obiecte de tipuri diferite sau cînd un operand este un obiect de un tip abstract, iar celălalt o dată de un tip predefinit.

De asemenea, apar situaţii în care se fac atribuiri în care intervin conversii de obiecte.

Situaţiile 2 şi 3 pot şi ele interveni cînd se transferă obiecte prin parametri sau se returnează un obiect la revenirea dintr-o funcţie.

Astfel de conversii nu pot fi aplicate automat, deoarece compilatorul C++ nu are definit modul de realizare al acestora, ca în cazul tipurilor predefinite.

Exemplu:

Considerăm tipul *complex* definit ca mai jos:

```
class complex {
     double real; double imag;
  public:
     complex(double x,double y)
     {
         real = x; imag = y;
     }

     complex operator +(complex a)
    /* returneaza un obiect de tip complex care este echivalent
     cu suma *this+a */
     {
         complex rtemp;
         rtemp.real = real+a.real;
         rtemp.imag = imag+a.imag;
         return rtemp;
     }
     ...
};
```

Supraîncărcarea operatorului + legalizează utilizarea expresiilor de forma:

```
z1+z2
```

unde:

z1 şi z2 - Sînt instanţieri ale clasei *complex*.

Expresia de mai sus se realizează prin apelul:

```
z1.operator+(z2)
```

În schimb, o expresie de forma:

```
z1+3
```

nu este legală, deoarece operatorul + nu este supraîncărcat decît numai pentru cazul cînd ambii operanzi sînt de tip complex.

Pentru a legaliza o astfel de expresie putem să mai adăugăm o funcţie membru care să supraîncarce operatorul + ca mai jos:

```
complex operator +(double a)
/* returneaza un obiect de tip complex echivalent cu
   suma *this+a */
{
 complex rtemp;
 rtemp.real = real+a;
 rtemp.imag = imag;
 return rtemp;
}
```

În felul acesta, expresia:

```
z1+3
```

este acceptată. La evaluarea ei se utilizează apelul:

```
z1.operator+(3)
```

Parametrul efectiv 3 se converteşte spre *double* (tipul parametrului formal care îi corespunde) şi apoi se execută corpul funcţiei operator + definită mai sus. Se observă că la evaluarea expresiei de mai sus s-a aplicat, în mod automat, o conversie implicită din tipul *int* în tipul *double*.

Funcţiile membru pentru supraîncărcarea operatorului + nu legalizează utilizarea expresiei:

```
3+z1
```

Aceasta se datorează faptului că, compilatorul evaluează această expresie cu ajutorul apelului:

```
3.operator+(z1)
```

care este eronat.

În general, dacă un operator binar este supraîncărcat printr-o funcţie membru nestatică, atunci utilizarea operatorului respectiv în expresii, cere ca primul operand să fie de tipul implementat de clasa pentru care funcţia respectivă este funcţie membru.

De aceea, pentru a legaliza o expresie de această formă vom folosi o funcţie prieten de felul celei de mai jos:

```
complex operator +(double a,complex z)
/* returneaza un obiect de tip complex echivalent cu suma a+z */
{
 complex rtemp;
 rtemp.real = real+a;
 rtemp.imag = imag;
 return rtemp;
}
```

sau mai simplu:

```
complex operator+(double a,complex z)
{
 return z+a;
}
```

în ipoteza că operatorul + este supraîncărcat pentru expresii în care operandul din stînga lui + este de tip *complex*. Cu alte cuvinte, pentru ca operatorul + să poată fi folosit pentru obiecte de tip *complex* sînt necesare 3 funcţii care să supraîncarce operatorul respectiv (2 funcţii membru nestatice şi o funcţie prieten). Clasa *complex* devine:

```
class complex {
     double real;
     double imag;
 public:
     complex(double x,double y)
     {
         real=x;
         imag=y;
     }
     complex operator+(complex z);
     // legalizeaza expresii de forma complex + complex

     complex operator+(double d);
     // legalizeaza expresii de forma complex+double
```

```
    friend complex operator+(double a,complex z);
    // legalizeaza expresii de forma double + complex
    ...
};
```

Pentru ceilalţi operatori sînt necesare supraîncărcări analoge.

Deşi această soluţie este simplă, ea conduce la un număr mare de funcţii *operator* pentru supraîncărcarea operatorilor binari.

Problema admiterii expresiilor cu operanzi de tipuri diferite pentru operatori binari poate fi rezolvată şi într-un alt mod şi anume prin definirea de conversii pentru date de tip abstract (obiecte).

Astfel, expresiile:

```
z1+3
```

şi

```
3+z1
```

se pot legaliza dacă se defineşte conversia datelor de tip *int* sau alt tip predefinit (double,float etc.) spre tipul *complex*. În acest caz, este nevoie ca operatorul + să fie supraîncărcat numai pentru cazul în care ambii operanzi sînt de tip *complex*. De asemenea, expresiile de mai sus devin legale dacă se defineşte conversia obiectelor de tip *complex* spre tipul *int* sau un alt tip predefinit.

În principiu, se pot defini astfel de conversii. Utilizatorul trebuie să aibă însă grijă să nu se ajungă la ambiguitate în aplicarea conversiilor de acest tip.

În general, se pot defini următoarele tipuri de conversii:

	Conversie	
	din	**în**
1.	tip predefinit	tip abstract
2.	tip abstract	tip predefinit
3.	tip abstract	tip abstract

Odată definite aceste conversii, ele se aplică în toate cele 3 situaţii,amintite la începutul acestui capitol şi anume, la evaluarea expresiilor şi la atribuiri, la transferul parametrilor şi returnarea rezultatelor la revenirea din funcţii.

Conversiile de felul celor de mai sus se numesc *conversii definite de utilizator* sau mai simplu *conversii utilizator*.

24.1. Conversia datelor de tipuri predefinite în date de tip abstract

Conversia unei date de un tip predefinit într-o dată de tip abstract se realizează în mod automat prin apelul unui constructor adecvat pentru tipul abstract respectiv.

În cazul clasei *complex* de mai sus, constructorul are doi parametri. El poate fi utilizat pentru a evalua expresiile:

```
z1+3
```

şi

```
3+z1
```

Astfel, expresiile:

```
z1+complex(3,0)
```

şi

```
complex(3,0)+z1
```

sînt legale şi la evaluarea lor se apelează constructorul clasei *complex* pentru a crea un obiect anonim de tip *complex*. Apoi, se apelează funcţia care supraîncarcă operatorul + pentru cazul în care ambii operanzi sînt de tip *complex*. O astfel de funcţie poate fi o funcţie membru sau o funcţie *prieten* a clasei *complex*.

Expresiile de mai sus sînt evaluate la fel ca şi expresia:

```
z1+z2
```

unde ambii termeni sînt de tip *complex*.

Exemplu:

Fie clasa:

```
class complex {
    double real;
    double imag;
  public:
    complex(double x,double y)
    {
        real = x;
```

```
        imag = y;
    }
    friend complex operator+(complex c1,
                             complex c2);
    //nu mai exista alte functii pentru supraincarcarea lui +
    ...
};
```

La evaluarea expresiei:

```
z1+z2
```

se realizează apelul:

```
operator+(z1,z2)
```

Transferul obiectelor *z1* şi *z2* se realizează cu ajutorul constructorului de copiere implicit, care este o copiere bit_cu_bit şi care este suficientă în cazul de faţă.

La evaluarea expresiei:

```
z1+complex(3,0)
```

se realizează apelul constructorului existent pentru clasa *complex* pentru a defini un obiect anonim cu partea reală egală cu 3 şi cu cea imaginară egală cu 0. Apoi,se apelează funcţia operator + sub forma:

```
operator+(z1,complex(3,0))
```

şi se transferă parametri, folosind constructorul de copiere implicit.

În mod analog, se evaluează expresia:

```
complex(3,0)+z1
```

De aici rezultă că, pentru a legaliza expresiile de forma:

```
z1+3
```

şi

```
3+z1
```

este necesar să se poată face apelurile:

```
operator+(z1,3)
```

şi respectiv:

```
operator+(3,z1)
```

La aceste apeluri se construieşte un obiect complex anonim care are

partea imaginară egală cu zero, iar apoi se apelează constructorul de copiere implicit pentru a realiza transferul de parametri în mod implicit. Aceasta se poate realiza folosind un constructor care are valoarea implicită zero pentru parametrul y.

Rezultă că prezenţa unui astfel de constructor permite tratarea de către compilator a expresiilor de forma:

z1+3

şi

3+z1

fără a mai fi nevoie de supraîncărcări corespunzătoare pentru operatorul + (adică supraîncărcarea pentru cazurile cînd un operand nu este de tip *complex*).

Menţionăm că, în exemplele date, se realizează şi o conversie de date predefinite şi anume la apelul constructorului, valoarea 3 de tip *int* se converteşte spre tipul *double*.

Exemplu:

```
class complex {
    double real;
    double imag;
 public:
    complex(double x,double y=0)
    {
        real=x;
        imag=y;
    }
    complex operator+(complex z);
    // nu mai există supraincarcari pentru operatorul +
    ...
};
```

Definiţia clasei *complex*, permite tratarea expresiilor de mai jos:

```
complex z1(1,2),z2(3,4);
...
z1+z2
...
z1+3
...
```

Prima expresie implică apelul:

```
z1.operator+(z2)
```

La acest apel, pointerul *this* se încarcă cu adresa lui *z1*(&z1), iar *z2* se copiază apelîndu-se constructorul de copiere implicit.

La cea de a doua expresie se foloseşte apelul:

```
z1.operator+(3)
```

La acest apel, pointerul *this* se încarcă ca mai sus, apoi se apelează constructorul clasei pentru a construi un obiect anonim care are partea imaginară egală cu 0. În final, se apelează constructorul de copiere implicit.

Să observăm că expresiile de forma:

```
3+z1
```

nu sînt legale. Într-adevăr, o astfel de expresie conduce la un apel de forma:

```
3.operator(z1)
```

care nu este acceptat de compilator. În acest caz, compilatorul nu mai apelează constructorul pentru a crea obiectul anonim de tip *complex* corespunzător lui 3. De aceea, pentru a admite toate variantele se recomandă să utilizăm pentru supraîncărcarea operatorului + o funcţie prieten, aşa cum s-a procedat într-un exemplu precedent. Acest lucru este valabil şi pentru ceilalţi operatori ai tipului *complex*.

Dacă un operator binar poate avea ca prim operand un operand de tip diferit decît cel pentru care se supraîncarcă, atunci el se va supraîncărca printr-o funcţie prieten. În caz contrar, se va supraîncărca printr-o funcţie membru nestatică. Avantajul funcţiilor membru nestatice faţă de funcţiile prieten constă în aceea că în cazul funcţiilor membru nestatice, primul operand se transferă cu ajutorul pointerului *this*. Acesta se aplică în mod implicit în corpul funcţiei membru.

Funcţiile membru nestatice se utilizează în mod obligatoriu cînd primul operand se modifică de către operatorul pe care-l supraîncarcă. Exemplu de astfel de operanzi sînt operatorii de atribuire.

Să observăm că prezenţa constructorului de mai sus, pentru clasa *complex*, nu intră în contradicţie cu funcţiile care supraîncarcă operatorul + pentru cazurile cînd nu sînt ambii operanzi de tip *complex*.

Într-adevăr, fie clasa *complex*, definită ca mai jos:

```
class complex {
      double real;
      double imag;
```

```
public:
    complex(double x,double y=0)
    {
        real=x;
        imag=y;
    }

    complex operator+(complex z);
    //realizeaza suma *this+z

    complex operator+(double a);
    //realizeaza suma *this+a

    friend complex operator+(double a,complex z);
    //realizeaza suma a+z
    ...
};
```

şi declaraţiile:

```
complex c(1,2);
int n;
...
```

Expresia:

```
c+n
```

se evaluează cu ajutorul apelului:

```
c.operator+(n)
```

La acest apel, se converteşte valoarea lui *n* spre double şi apoi se apelează, în mod obişnuit, funcţia membru care supraîncarcă operatorul +, de antet:

complex operator+(double)

În mod analog, expresia:

```
n+c
```

se evaluează cu ajutorul apelului:

```
operator+(n,c)
```

La acest apel se converteşte valoarea lui *n* spre *double*, apoi se apelează funcţia prieten care supraîncarcă operatorul +.

În ambele cazuri, nu intervine conversia din tipul predefinit *double* spre

tipul *complex*.

Conversia dintr-un tip predefinit într-un tip abstract se realizează numai dacă nu se poate selecta o funcţie membru sau prieten a cărei parametri să nu necesite o astfel de conversie.

Ca o concluzie de reţinut este faptul că, o conversie dintr-un tip predefinit într-un tip abstract se realizează de către un constructor al clasei prin care se implementează tipul abstract respectiv. Acest constructor are un parametru care trebuie să fie de un tip predefinit care se poate converti spre tipul abstract. Constructorul poate avea şi alţi parametri, dar aceştia vor fi toţi impliciţi. O conversie dintr-un tip predefinit într-un tip abstract poate fi precedată de alte conversii care se realizează între tipuri predefinite. De exemplu, mai sus, constanta 3 a fost convertită din tipul *int* în tipul *double* şi apoi în tipul *complex*.

Conversiile de acest tip au o prioritate redusă. Ele se aplică atunci cînd supraîncărcarea operatorilor nu se aplică în mod direct sau folosind conversii predefinite. De exemplu, atribuirea:

```
z=n
```

unde *z* este de tip complex, iar *n* de tip predefinit (de exemplu *int*) se realizează apelînd funcţia membru nestatică care supraîncarcă operatorul de atribuire pentru clasa *complex* şi care are antetul:

```
complex& operator = (double a)
```

dacă o astfel de funcţie există. În caz contrar,se aplică constructorul care converteşte pe *n* într-un obiect anonim de tip *complex* şi apoi se face atribuirea, folosind funcţia membru nestatică care supraîncarcă operatorul de atribuire pentru clasa *complex* şi de antet:

```
complex& operator = (complex z1)
```

dacă o astfel de funcţie există. În caz contrar, se face o copiere bit_cu_bit.

Conversiile definite de utilizator pe baza aplicării automate a constructorului se numesc conversii utilizator *implicite*.

Obiectele construite prin conversii utilizator implicite se distrug automat cînd nu mai este nevoie de ele.

Conversiile utilizator implicite sînt adesea utile deoarece, aşa cum s-a arătat mai sus, permit reducerea substanţială a funcţiilor pentru supraîncărcarea operatorilor. Cu toate acestea, aplicarea conversiilor utilizator implicite trebuie evaluată în legătură cu eficienţa lor faţă de utilizarea funcţiilor de supraîncărcare a operatorilor pe care le elimină.

De asemenea, la aplicarea conversiilor utilizator implicite va trebui să

avem în vedere posibilităţile de apariţie a ambiguităţilor.

Fie, de exemplu, funcţia *f* supraîncărcată ca mai jos:

```
void f(cl1 x);
void f(cl2 x);
```

unde:

cl1 şi cl2 - Sînt două clase care au constructori de prototipuri:

```
cl1(double);
```

şi

```
cl2(double);
```

Aceşti constructori, avînd un singur parametru, realizează conversii utilizator implicite din *double* spre tipul abstract *cl1*, respectiv *cl2*.

Un apel de forma:

```
f(3.14159265);
```

este eronat, deoarece nu este definită în mod unic conversia utilizator implicită. În acest caz este necesară aplicarea explicită a constructorului:

```
f(cl1(3.14159265));
```

sau

```
f(cl2(3.14159265));
```

Să observăm că, dacă funcţia *f* nu este supraîncărcată, de exemplu este valabil numai primul prototip, atunci apelul iniţial de mai sus este corect şi se va realiza în mod automat conversia din *double* spre *cl1*.

Observaţie:

Fie *cl* o clasă care se defineşte ca mai jos:

```
class cl {
    ...
 public:
    cl();                    // constructor implicit
    cl(tip_predefinit);  /*constructor pentru conversii
                                utilizator implicite */
    cl(...);
    cl(const cl&);          // constructor de copiere
    ...
    cl& operator=(cl);  /* supraincarcarea operatorului de
                                atribuire */
```

```
    ...
};
```

Constructorii şi funcţia de supraîncărcare a operatorului de atribuire se utilizează ca mai jos:

```
cl ob1,ob2;  // se aplica constructorul implicit
tip_predefinit t;
...
    cl ob3(t);      /* se aplica constructorul pentru conversii
                       utilizator implicite */
    cl ob4=t;       /* se aplica constructorul pentru conversii
                       implicite; se obtine un obiect anonim de
                       tip cl, apoi acesta se copiaza bit_cu_bit
                       in obiectul ob4; dupa copiere se distruge
                       obiectul anonim */
    cl ob5(ob4);    // se aplica constructorul de copiere
    cl ob6 = ob4;   // se aplica constructorul de copiere
    ob1 =ob3;       /*se apeleaza functia care supraincarca
                       operatorul = */
    ob2=t;          /* se aplica constructorul pentru conversii
                       utilizator implicite si se creeaza un
                       obiect anonim de tip cl, apoi acesta se
                       copiaza bit_cu_bit in obiectul ob2; dupa
                       copiere obiectul anonim se distruge */

    void f(cl);     // prototipul functiei f
    ...
    f(ob1);         /*se aplica constructorul de copiere pentru a
                       atribui parametrului formal al functiei f,
                       valoarea parametrului efectiv ob1 */
    ...
    f(t);           /* se apeleaza constructorul pentru conversii
                       implicite; se obtine un obiect anonim de
                       tip cl, apoi acesta se copiaza bit_cu_bit in
                       parametrul formal al functiei f; dupa copie-
                       re se distruge obiectul anonim */
```

Exerciţii:

24.1 Să se definească tipul abstract *complex* supraîncărcînd operatorii pentru operaţii cu numere complexe cu un număr minim de funcţii.

Tipul *complex* a fost implementat în exerciţiul 23.1. folosind cîte 3 funcţii pentru fiecare operator binar.

În acest exerciţiu,fiecare operator binar va fi supraîncărcat printr-o singură funcţie, care este o funcţie prieten. Excepţie de la această regulă o constituie operatorul de împărţire pentru care se definesc două funcţii: una care este funcţie prieten şi se apelează la împărţirea de numere complexe cînd ambii operanzi sînt numere complexe sau cînd împărţitorul este complex, iar deîmpărţitul nu este complex şi una care este funcţie membru nestatică şi se aplică atunci cînd împărţitorul nu este complex. În acest caz, este mai eficient ca împărţirea să nu se facă prin metoda generală de împărţire a două numere complexe.

Celelalte funcţii membru sînt aceleaşi ca în exerciţiul 23.1.

FIŞIERUL BXXIV1.H

```
class complex {
    double real;
    double imag;

    static int citdouble(char *s,double& d);
   /* - afiseaza s;
       - citeste un numar si-l atribuie datei referite de d;
       - returneaza:
              0 - la sfirsit de fisier;
              1 - altfel.
     */
 public:     // constructor
    complex(double x=0,double y=0);
    /* se utilizeaza pentru:
           - instantieri de obiecte fara parametri (real=imag=0);
           - instantieri de obiecte cu 1-2 parametri;
           - conversie utilizator implicita dintr-un tip predefinit in tipul
             complex.
     */

    // modulul numarului complex
    double modul();

    // argumentul numarului complex
    double arg();

    // citeste un numar complex
    int citcomplex(char *s);
    /* - afiseaza s;
       - citeste componentele numarului complex;
```

```
   - returneaza:
         0 - la sfirsit de fisier;
         1 - altfel.
  */

 // afiseaza un numar complex
 void afiscomplex(char *f);
/* - afiseaza componentele numarului complex;
   - se utilizeaza formatul spre care pointeaza f sau un format
     standard daca f pointeaza spre sirul vid. */

// incrementeaza partea imaginara
void ipi();

// decrementeaza partea imaginara
void dpi();
/* - obiectul curent se noteaza cu z;
   - obiectul rezultat se noteaza cu r. */

// operatori unari

// negativare
complex operator -();   // r = -z

// incrementeaza partea reala
complex operator ++();

// decrementeaza partea reala
complex operator --();

// radacina patrata
complex operator !();

// operatori binari

// adunare r = z1+z2
friend complex operator +(complex z1,
                          complex z2);

// scadere r = z1-z2
friend complex operator -(complex z1,
                          complex z2);
```

```
    // inmultire r = z1*z2
    friend complex operator *(complex z1,
                              complex z2);

    // impartire r = z1/z2
    friend complex operator /(complex z1,
                              complex z2);

    // r = z/d
    complex operator /(double d);

    // ridicare la putere r = z**i
    complex operator ^(int i);

    // test de egalitate z1 == z2
    friend int operator == (complex z1,
                            complex z2);

    // returneaza partea reala
    double retreal();

    // returneaza partea imaginara
    double retimag();
};
```

Funcţiile membru se definesc în fişierul de mai jos, de extensie CPP.

FIŞIERUL BXXIV1

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO__H
#endif
#ifndef __MATH_H
#include <math.h>
#define __MATH__H
#endif
#ifndef __PI
#define PI 3.14159265358979
#define __PI
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB__H
#endif
```

```
#ifndef __STRING_H
#include <string.h>
#define __STRING_H
#endif
#ifndef __COMPLEX_H
#include "BXXIV1.H"
#define __COMPLEX_H
#endif

inline complex::complex(double x,double y)
/* constructor utilizat la initializari si conversia utilizator implicita
   dintr-un tip predefinit in tipul complex */
{
 real = x;
 imag = y;
}

inline double complex::modul()
// returneaza modulul numarului complex
{
 return sqrt(real*real+imag*imag);
}

double complex::arg()
// returneaza argumentul numarului complex
{
 if(real==0&&imag==0)
    return 0.0;
 if(imag==0)
    if(real > 0)
        return 0.0;
    else
        return PI;
 if(real==0)
    if(imag > 0)
        return PI/2;
    else
        return (3*PI/2);
 double a = atan(imag/real);
 if(real < 0)
    return PI+a;
 if(imag < 0)
    return 2*PI+a;
```

```
        return a;
}

int complex::citdouble(char *s,double& d)
/* - afiseaza s;
   - citeste un numar si-l atribuie datei referite de d;
   - returneaza:
          0 - la sfirsit de fisier;
          1 - altfel.
*/
{
 char t[255];

 for(;;){
     if(s && *s)
         printf(s);
     if(gets(t)==0)
         return 0;
     if(sscanf(t,"%lf",&d)==1)
         break;
     printf("nu s-a tastat un numar\n");
     printf("se reia citirea\n");
 }
 return 1;
}

int complex::citcomplex(char *s)
/* - afiseaza s daca nu este vid; daca este vid se afiseaza texte standard:
        - Partea reala si Partea imaginara;
   - citeste componentele obiectului complex curent;
   - returneaza:
          0 - la sfirsit de fisier;
          1 - altfel.
*/
{
 if(s && *s) {     //sirul s nu este vid;
     //se citesc componentele obiectului curent
     for(;;) {
         printf(s);
         int i;
         double d;
         if((i=scanf("%lf",&d)) == EOF)
             return 0;
```

```
    if(i!=1){
        printf("nu s-a tastat un numar pentru\
                partea reala\n");
        printf("se reia citirea de la\
                inceput\n");
        fflush(stdin);
        continue;
    }
    else
        real = d;
    if((i=scanf("%lf",&d))==EOF)
        return 0;
    if(i!=1){
        printf("nu s-a tastat un numar\n");
        printf("pentru partea imaginara\n");
        printf("se reia citirea de la\
                inceput\n");
        fflush(stdin);
    }
    else{
        imag=d;
        fflush(stdin);
        return 1;
    }
  } // sfirsit for
} // sfirsit if(s && *s)
```

// s pointeaza spre sirul vid sau este pointerul zero

```
if(complex::citdouble("Partea reala=",real) == 0)
    return 0;
if(complex::citdouble("Partea imaginara=",
  imag) == 0)
    return 0;
return 1;
}

void complex::afiscomplex(char *f)
```

/* - afiseaza componentele obiectului curent cu formatul f;
 - daca f reprezinta sirul vid, se utilizeaza un format standard */

```
{
  if(f && *f)     // formatul de afisare nu este vid
```

```
        printf(f,real,imag);
    else        // format standard
        printf("%g+i*(%g)\n",real,imag);
}

inline void complex::ipi()
/* incrementeaza partea imaginara a obiectului complex curent */
{
 imag++;
}

inline void complex::dpi()
/* decrementeaza partea imaginara a obiectului curent */
{
 imag--;
}

complex complex::operator-()   // r = -z
{
 complex r;

 r.real = -real;
 r.imag = -imag;
 return r;
}

inline complex complex::operator++()
/* incrementeaza partea reala a obiectului curent */
{
 real++;
 return *this;
}

inline complex complex::operator--()
/* decrementeaza partea reala a obiectului complex curent */
{
 real --;
 return *this;
}
```

```
complex complex::operator !()
/* calculeaza radacina patrata din obiectul complex curent */
{
 complex r;   // real=imag=0
 double d;

 if((d=modul())==0)
 return r;   // obiect complex nul
 double alfa=arg();  // argumentul obiectului complex
 d = sqrt(d);  // radacina patrata din modul
 alfa = alfa/2;
 r.real = d*cos(alfa);
 r.imag = d*sin(alfa);
 return r;
}

complex operator+(complex z1,complex z2)  // r = z1+z2
{
 complex r;

 r.real = z1.real+z2.real;
 r.imag = z1.imag+z2.imag;
 return r;
}

complex operator -(complex z1,complex z2)  // r = z1-z2
{
 complex r;

 r.real = z1.real-z2.real;
 r.imag = z1.imag-z2.imag;
 return r;
}

complex operator *(complex z1,complex z2)  // r = z1*z2
{
 complex r;

 r.real = z1.real*z2.real-z1.imag*z2.imag;
 r.imag = z1.real*z2.imag+z1.imag*z2.real;
 return r;
}
```

```
complex operator /(complex z1,complex z2)
// r = z1/z2; r=0 daca z2=0
{
 complex r;  // real=imag=0
 double d = z2.real*z2.real+z2.imag*z2.imag;

 if(d==0)  // z2 = 0
    return r;
 r.real = (z1.real*z2.real+z1.imag*z2.imag)/d;
 r.imag = (z1.imag*z2.real-z1.real*z2.imag)/d;
 return r;
}

complex complex::operator /(double d)
// r = z/d; r=0 daca d=0
{
 complex r;  // real=imag=0

 if(d==0)
    return r;
 r.real = real/d;
 r.imag = imag/d;
 return r;
}

complex complex::operator ^(int i)
// r = z**i; r=0 daca z=0
{
 complex r;  // real=imag=0
 double d=modul();

 if(d==0)
    return r;
 int n=i < 0 ? -i : i;   // n = abs(i)
 if(n==0)      // z**0=1
    return 1;
 if(i==1)      // z**1=z
    return *this;
 if(i==-1)     // z**(-1) = 1/z
    r = *this;
 if(n > 1){
    d = pow(d,(double)n);  // d=d**n
```

```
      double a = arg()*n;  //argument*n

    // reducere la primul cerc

      long x = a/PI;
      a -= x*PI;
      r.real = d*cos(a);
      r.imag = d*sin(a);
  }
  if(i < 0)  //z**i=1/z**n; r=z**n
      r = 1/r;  /* operatorul / este definit pentru
                   complex/complex; se aplica conversia implicita */
  return r;
}

inline int operator == (complex z1,complex z2)
/* returneaza:
     1 - daca z1=z2;
     0 - altfel.
*/
{
  return z1.real == z2.real && z1.imag == z2.imag;
}

inline double complex::retreal()
// returneaza partea reala a obiectului complex curent
{
  return real;
}

inline double complex::retimag()
// returneaza partea imaginara a obiectului complex curent
{
  return imag;
}
```

24.2 Să se scrie un program care citeşte valorile variabilelor *n*, *a* şi *b*, calculează şi afişează rădăcinile ecuaţiei binome:

a*x**n+b = 0

unde:

n - este un întreg pozitiv;

a,b - sînt numere complexe.

Rădăcinile ecuaţiei binome se obţin calculînd rădăcinile de ordinul *n* din numărul complex -b/a, dacă n > 1. Pentru n=1 ecuaţia de mai sus este de grad 1 şi are o singură soluţie:

x = -b/a

Rădăcinile de ordinul *n* dintr-un număr complex *z* se determină din forma trigonometrică a acestuia.

Fie *r* şi *alfa*, modulul şi respectiv argumentul numărului complex *z*. Atunci rădăcina a k-a de ordinul *n* din numărul complex *z* se calculează astfel:

rad_k=rn*(cos(beta_k)+i*sin(beta_k))

unde:

rn - Este rădăcina de ordinul *n* din r;

beta_k = alfa/n+k(2*PI/n)*

În aceste relaţii, *k* variază de la 0 la n-1.

PROGRAMUL BXXIV2

```
#include "BXXIV1.CPP"
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#include "BXX4.CPP"   // rcit_int
#include "BXX5.CPP"   // rcit_int_lim

main()
/* - citeste pe: n,a si b;
   - calculeaza si afiseaza radacinile ecuatiei binome:
        a*x**n+b = 0
     unde:
        n  - intreg pozitiv;
        a,b - numere complexe
*/
{
 int k,n;
 complex a,b,x,z;

 for(;;){
    printf("Actionati o tasta pentru a\
```

```
continua\n");
getch();
if(rcit_int_lim("gradul ecuatiei n=",
                1,32767,n)==0)
   exit(0);
printf("coeficientul lui x\n");
if(a.citcomplex("")==0)
   exit(1);  //EOF
printf("termenul liber\n");
if(b.citcomplex("")==0)
   exit(1);  //EOF
if(a==0&&b==0){
   printf("ecuatie nedeterminata\n");
   continue;   //se trece la alta ecuatie
}
if(a==0){
   printf("ecuatia nu are solutie\n");
   continue;   //se trece la alta ecuatie
}
z=-b/a;
if(n==1){    //ecuatie de gradul 1
   printf("ecuatie de gradul 1\n");
   z.afiscomplex("x=%g+i(%g)\n");
   continue;  //se trece la alta ecuatie
}

//ecuatie de grad mai mare ca 1
//se determina modulul si argumentul numarului complex z

double r = z.modul();
double alfa = z.arg();
double rn = pow(r,1.0/n); //radacina de ordinul n din r
alfa = alfa/n;
double doipipen = 2*PI/n;

//calculeaza si afiseaza cele n radacini

for(k=0;k < n;k++){
   double beta = alfa+k*doipipen;
   x=complex(rn*cos(beta),rn*sin(beta));
   printf("x%d=",k);
   x.afiscomplex("");
   if((k+1)%10==0){
```

```
            printf("Actionati o tasta pentru a\
                    continua\n");
            getch();
         }
      }
      printf("\n");
   }
}
```

24.3 Să se scrie un program care rezolvă ecuaţii de gradul 2 cu coeficienţi complecşi.

PROGRAMUL BXXIV3

```
#include <conio.h>
#include "BXXIV1.CPP"

main()
/* rezolva ecuatii de gradul 2 cu coeficienti complecsi de forma:
      a*x*x+b*x+c = 0
*/
{
 complex a,b,c,x;

 for(;;){
    printf("Actionati o tasta pentru a\
          continua\n");
    getch();
    if(a.citcomplex("a=")==0)
       exit(0);  //EOF
    if(b.citcomplex("b=")==0)
       exit(1);  //EOF
    if(c.citcomplex("c=")==0)
       exit(1);  //EOF
    if(a==0 && b==0 && c==0){
       printf("ecuatie nedeterminata\n");
       continue;
    }
    if(a==0 && b==0){
       printf("ecuatia nu are solutie\n");
       continue;
    }
    if(a==0){   //ecuatie de gradul 1
       printf("ecuatie de gradul 1\n");
```

```
            x = -c/b;
            x.afiscomplex("x=%g+i*(%g)\n");
            continue;
        }

        //a != 0 - ecuatie de gradul 2

        complex doia = 2*a;
        complex delta = !(b*b-4*a*c);
        x = (-b+delta)/doia;
        x.afiscomplex("x1=%g+i*(%g)\n");
        x=(-b-delta)/doia;
        x.afiscomplex("x2=%g+i*(%g)\n");
    }
}
```

24.4 Să se implementeze tipul abstract *grad* pentru lucrul cu grade sexagesimale.

Un grad sexagesimal are 60 de minute sexagesimale, iar un minut sexagesimal are 60 de secunde sexagesimale.

Tipul abstract *grad* permite operaţii cu grade sexagesimale:

- adunare;
- scădere;
- înmulţire şi împărţire în care un operand este de tip *grad* şi unul este de tip predefinit.

De asemenea, clasa *grad* are funcţii membru care permit citiri şi afişări de obiecte de tip *grad*.

Clasa are 4 date membru protejate:

- *sgrad*;
- *smin*;
- *ssec*;

şi

- *totsec*

Între aceste date există relaţia:
totsec = sgrad*3600+smin*60+ssec

Funcţia membru *grminsec*, determină valorile componentelor *sgrad*,

smin şi *ssec* din valoarea datei membru *totsec*.

Există doi constructori, unul care permite iniţializarea obiectelor cu valoarea variabilei *totsec*, iar unul care permite iniţializarea obiectelor cu valori pentru *sgrad*, *smin* şi *ssec*.

În mod analog, există 2 funcţii membru pentru a citi valori numai pentru componenta *totsec* sau pentru a citi valori exprimate în grade, minute şi secunde.

FIŞIERUL BXXIV4.H

```
class grad {
      long totsec;
      long sgrad;
      int smin;
      int ssec;

      void grminsec();
      /* converteste din total secunde in grade, minute si secunde */

   public:
      grad(long s=0);
      /* - permite declarari de obiecte care se initializeaza dindu-se
           numai secundele;
         - permite conversie utilizator implicita. */

      grad(long gr,int min,int sec);
      /* initializeaza obiecte dindu-se gradele, minutele si secundele */

      int citsec();
      /* - citeste un intreg de tip long care reprezinta secundele pentru
            obiectul curent;
         - returneaza zero la intilnirea sfirsitului de fisier si 1 altfel. */

      int citgms();
      /* - citeste trei intregi care reprezinta grade, minute si secunde si
            care se atribuie respectiv datelor sgrad, smin si ssec ale
            obiectului curent;
         - returneaza zero la intilnirea sfirsitului de fisier si 1 altfel. */

      void afissec(char *f);
      /* afiseaza, folosind formatul f, valoarea variabilei totsec pentru
         obiectul curent; daca f pointeaza spre sirul vid, atunci se
         utilizeaza un format standard */
```

```
void afisgrad(char *f);
/* afiseaza cu formatul f, valoarea variabilelor sgrad, smin si ssec;
   daca f pointeaza spre sirul vid, atunci se utilizeaza un format
   standard */

friend grad operator +(grad g1,grad g2);
// returneaza suma g1+g2

friend grad operator -(grad g1,grad g2);
// returneaza diferenta g1-g2

grad operator-();
// returneaza obiectul curent negativat

grad operator *(double a);
// returneaza produsul g*a; g este obiectul curent

friend grad operator *(double a,grad g2);
// returneaza produsul a*g2

grad operator /(double a);
// returneaza citul g/a; g este obiectul curent

grad& operator += (grad g2);
// g = g+g2; g este obiectul curent

grad& operator -= (grad g2);
// g = g-g2; g este obiectul curent

long retsec();
// returneaza valoarea lui totsec pentru obiectul curent

long retg();
// returneaza gradele pentru obiectul curent

int retm();
// returneaza minutele pentru obiectul curent

int rets();
// returneaza secundele pentru obiectul curent

friend int operator == (grad g1,grad g2);
// returneaza 1 daca g1=g2 si zero altfel
```

```
      friend int operator < (grad g1,grad g2);
      // returneaza 1 daca g1 < g2 si zero altfel
      friend int operator <= (grad g1, grad g2);
      // returneaza 1 daca g1 <= g2 si zero altfel
};
```

Observaţie:

Modul în care au fost supraîncărcaţi operatorii * şi / asigură protecţie împotriva înmulţirilor sau împărţirilor cu ambii operanzi de tip *grad*.

Mai jos, se definesc funcţiile membru în fişierul de extensie *cpp*.

Tot în acest fişier se apelează două funcţii care permit citirea întregilor de tip *long*. Ele sînt asemănătoare cu funcţiile *rcit_int* şi *rcit_int_lim* definite în exerciţiile 20.4 şi respectiv 20.5. Numele lor se obţin din numele funcţiilor respective schimbînd cuvîntul *int* cu *long*.

FIŞIERUL BXXIV4

```
#ifndef __GRAD_H
#include "BXXIV4.H"
#define __GRAD_H
#endif

void grad::grminsec()
/* converteste din totalul secunde in grade, minute si secunde */
{
 sgrad = totsec/3600;
 int rest = totsec%3600;
 smin = rest/60;
 ssec = rest%60;
}

inline grad::grad(long s)
/* initializeaza totsec cu valoarea lui s si apoi si celelalte componente in
  mod corespunzator */
{
 totsec = s;
 grminsec();
}
```

```
inline grad::grad(long g,int m,int s)
/* asigura instantierea obiectelor cu valori pentru grade, minute si
   secunde */
{
 totsec = g*3600+m*60+s;
 grminsec();
}

int grad::citsec()
/* - citeste un intreg de tip long care reprezinta secunde pentru obiectul
   curent;
 - returneaza:
        0 - la intilnirea sfirsitului de fisier;
        1 - altfel.
*/
{
 long s;

 if(rcit_long_lim("Masura in secunde=",
          -2147483648,2147483647,s)==0)
    return 0;
    totsec=s;
    grminsec();
    return 1;
}

int grad::citgms()
/* - citeste trei intregi care reprezinta grade, minute si secunde si care se
   atribuie respectiv datelor sgrad, smin si ssec ale obiectului curent;
 - returneaza:
        0 - la intilnirea sfirsitului de fisier;
        1 - altfel. */
{
 long g;

 if(rcit_long_lim("grade=",
     -2147483648,2147483647,g)==0)
    return 0;

 int m;

 if(rcit_int_lim("minute=",-60,60,m)==0)
    return 0;
```

```
 int s;
 if(rcit_int_lim("secunde=",-60,60,s)==0)
    return 0;
 totsec = g*3600+m*60+s;
 grminsec();
 return 1;
}
```

```
void grad::afissec(char *f)
```

/* afiseaza, folosind formatul f, valoarea variabilei totsec; daca f pointeaza spre sirul vid, atunci se utilizeaza un format standard */

```
{
 if(f&&*f)
    printf(f,totsec);
 else
    printf("masura in secunde=%ld\n",totsec);
}
```

```
void grad::afisgrad(char *f)
```

/* afiseaza, folosind formatul f, masura unghiului in grade, minute si secunde; daca f pointeaza spre sirul vid, atunci se utilizeaza un format standard */

```
{
 if(f&&*f)
    printf(f,sgrad,smin,ssec);
 else
    printf("grade=%ld\tminute=%d\tsecunde=%d\n",
           sgrad,smin,ssec);
}
```

```
grad operator +(grad g1,grad g2)
```

// returneaza suma g1+g2

```
{
 grad r;
 r.totsec=g1.totsec+g2.totsec;
 r.grminsec();
 return r;
}
```

```
grad operator -(grad g1,grad g2)
```

// returneaza diferenta g1-g2

```
{
 grad r;
```

```
 r.totsec = g1.totsec-g2.totsec;
 r.grminsec();
 return r;
}

grad grad::operator-()
// returneaza negativul obiectului curent
{
 grad r;

 r.totsec=-totsec;
 r.grminsec();
 return r;
}

grad grad::operator*(double a)
// returneaza g*a; g este obiectul curent
{
 grad r;

 r.totsec=totsec*a;
 r.grminsec();
 return r;
}

grad operator *(double a,grad g2)   // returneaza a*g2
{
 return g2*a;
}

grad grad::operator/(double a)
// returneaza g/a; g este obiectul curent
{
 grad r;

 r.totsec=totsec/a;
 r.grminsec();
 return r;
}
```

```
grad& grad::operator += (grad g2)
// g=g+g2; g este obiectul curent
{
 totsec += g2.totsec;
 grminsec();
 return *this;
}

grad& grad::operator -= (grad g2)
// g = g-g2; g este obiectul curent
{
 totsec -= g2.totsec;
 grminsec();
 return *this;
}

inline long grad::retsec()
// returneaza masura unghiului in secunde
{
 return totsec;
}

inline long grad::retg()
// returneaza gradele pentru obiectul curent
{
 return sgrad;
}

inline int grad::retm()
// returneaza minutele pentru obiectul curent
{
 return smin;
}

inline int grad::rets()
// returneaza secundele pentru obiectul curent
{
 return ssec;
}
```

```
inline int operator == (grad g1,grad g2)
// returneaza 1 daca g1=g2 si zero altfel
{
 return g1.totsec==g2.totsec;
}
inline int operator < (grad g1,grad g2)
// returneaza 1 daca g1 < g2 si zero altfel
{
 return g1.totsec < g2.totsec;
}

inline int operator <= (grad g1,grad g2)
// returneaza 1 daca g1 <= g2 si zero altfel
{
 return g1.totsec <= g2.totsec;
}
```

24.5 Să se scrie o funcţie care afişează caracterele unui tablou şi citeşte un întreg de tip *long*. Funcţia are doi parametri:

- text: tablou unidimensional de tip caracter şi care are aceeaşi utilizare ca în exerciţiile 6.3, 8.2 şi 20.4;
- x: parametru de tip referinţă la întregi de tip *long*; numărul citit se păstrează în zona de memorie alocată parametrului efectiv corespunzător lui x.

FUNCŢIA BXXIV5

```
long rcit_long(char text[],long& x)
/* - afiseaza sirul de caractere spre care pointeaza text;
   - citeste un intreg si-l pastreaza in zona de memorie alocata
    parametrului efectiv corespunzator lui x;
   - returneaza:
         0 - la intilnirea sfirsitului de fisier;
         1 - altfel. */
{
 char t[255];

 for(;;){
    printf(text);
    if(gets(t)==0)
       return 0;
    if(sscanf(t,"%ld",&x)==1)
```

```
            return 1;
         printf("Nu s-a tastat un intreg\n");
      }
   }
```

24.6 Să se scrie o funcţie care citeşte un întreg de tip *long* care aparţine unui interval dat.

Această funcţie este analogă cu funcţiile definite în exerciţiile 8.3 şi 20.5. Funcţiile respective citesc întregi de tip *int* care aparţin unui interval dat.

FUNCŢIA BXXIV6

```
long rcit_long_lim(char text[],long inf,
                   long sup,long& rlong)
/* - afiseaza sirul de caractere spre care pointeaza text;
   - citeste un intreg de tip long ce apartine intervalului [inf,sup] si-l
     pastreaza in zona de memorie alocata parametrului efectiv
     corespunzator lui rlong;
   - returneaza:
        0 - la intilnirea sfirsitului de fisier;
        1 - altfel.
*/
{
 for(;;){
      if(rcit_long(text,rlong)==0)
         return 0;  // s-a intilnit EOF
      if(rlong >= inf && rlong <= sup)
         return 1;
      printf("intregul tastat nu apartine\
              intervalului:");
      printf("[%ld,%ld]\n",inf,sup);
      printf("se reia citirea\n");
 }
}
```

24.7 Să se scrie un program care citeşte măsura unei laturi a unui triunghi, precum şi măsurile, în grade sexagesimale, ale unghiurilor alăturate laturii respective şi pe baza acestor date calculează şi afişează măsurile celorlalte elemente ale triunghiului.

Programul rezolvă succesiv problema enunţată pentru mai multe triunghiuri şi se opreşte la întîlnirea sfîrşitului de fişier.

Fie *a* măsura laturii citite şi B şi C măsurile în grade sexagesimale ale unghiurilor alăturate lui *a*. Atunci A = 180-B-C şi a/sinA = b/sinB = c/sinC.

Măsurile laturilor b şi c se găsesc conform relaţiilor:

b = (a/sinA)*sinB

şi

c = (a/sinA)*sinC

Pentru calculul funcţiei sinus este nevoie ca măsura unghiului să fie în radiani.

Trecerea de la grade sexagesimale în radiani se poate face cu ajutorul relaţiei:

r=(PI/180)*n

unde:

r - Este măsura unghiului în radiani.

PI - Are valoarea 3.14159265358979.

n - Este măsura unghiului in grade sexagesimale.

Dacă măsura unghiului se dă în secunde, atunci relaţia de mai sus devine:

r = (PI/(180*3600))*s

unde prin *s* s-a notat măsura unghiului în secunde. Aceasta se mai scrie:

r=(PI/648000)*s

PROGRAMUL BXXIV7

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <math.h>
#include "BXX4.CPP"      // rcit_int
#include "BXX5.CPP"      // rcit_int_lim
#include "BXXIV5.CPP"    // rcit_long
#include "BXXIV6.CPP"    // rcit_long_lim
#define PI 3.14159265358979
#include "BXXIV4.CPP"    // implementeaza tipul grad

main()
/* - citeste:
        - masura laturii a;
        - masurile in grade sexagesimale ale unghiurilor B si C, alaturate
          laturii a;
      - calculeaza si afiseaza masurile celorlalte elemente ale
        triunghiului */
```

```
{
 double a;
 grad A,B,C;
 double factor = PI/648000;
 char er[] = "s-a tastat EOF\n";
 char erunghi[]="Masura unghiului nu este\
             in (0,180)\n";

 for(;;){

     // citeste masura laturii a
     for(;;){
        char t[255];
        printf("masura laturii a=");
        if(gets(t)==0)
            exit(0);  // EOF
        if(sscanf(t,"%lf",&a) == 1 && a > 0)
            break;
        printf("nu s-a tastat un numar pozitiv\n");
     }
     for(;;){
         // citeste masura unghiului B
         // se tasteaza grade minute si secunde
         for(;;){
            printf("B=");
            if(B.citgms()==0){
               printf(er);
               exit(1);
            }
            if(0 < B && B < 180*3600L)
               break;
            printf(erunghi);
         }
         for(;;){
            printf("C=");
            if(C.citgms()==0){
               printf(er);
               exit(1);
            }
            if(0 < C && C < 180*3600L)
               break;
            printf(erunghi);
```

```
        }
        A = 180*3600L-(B+C);
        if(A <= 0){
           printf("Suma B+C depaseste 180 de\
                   grade\n");
           printf("Se reia citirea masurilor\
                   unghiurilor\n");
           continue;
        }
        break;
     } // sfirsit citirea datelor pentru un triunghi

     // afiseaza datele citite
     printf("\n\n Masura laturii a = %g\n",a);
     B.afisgrad("Masura unghiului B: grade=%ld\
              minute=%d secunde = %d\n");
     C.afisgrad("Masura unghiului C: grade=%ld\
              minute=%d secunde = %d\n");

     // afiseaza datele calculate
     printf("\n\n Date calculate\n");
     A.afisgrad("Masura unghiului A: grade=%ld\
              minute=%d secunde = %d\n");
     double f=a/sin(A.retsec()*factor);
     double b=f*sin(B.retsec()*factor);
     double c = f*sin(factor*C.retsec());
     printf("Masura laturii b = %g\n",b);
     printf("Masura laturii c = %g\n",c);
  }
}
```

24.2. Conversia dintr-un tip abstract într-un tip predefinit

Conversia datelor de tip abstract spre o dată de tip predefinit se realizează pe baza supraîncărcării operatorului *cast* (operatorul de forţare a tipului). Acesta se supraîncarcă folosind o funcţie membru nestatică care are un antet specific:

nume_clasa::operator nume_tip_predefinit()

Lipsa parametrului rezultă din faptul că operatorul *cast* este unar.

Funcţia care supraîncarcă operatorul *cast* se apelează pentru un obiect

care este o instanţiere a clasei *nume_clasa* şi rezultatul apelului este o dată de tip *nume_tip_predefinit* şi a cărei valoare este definită prin execuţia corpului funcţiei respective.

Operatorul *cast* obişnuit se aplică de către programator la un operand pentru a impune o anumită conversie dintr-un tip predefinit în altul tot predefinit.

Am văzut că, în acest scop, se pot utiliza două formate:

(*tip*)*expresie*

sau

tip(*expresie*)

Operatorul *cast* supraîncărcat pentru o clasă oarecare se aplică pentru o instanţiere a clasei respective fie explicit, fie implicit.

La aplicarea explicită a operatorului *cast* supraîncărcat se utilizează formate similare cu cele indicate mai sus:

(*nume_tip_predefinit*)*obiect*

sau

nume_tip_predefinit(*obiect*)

În lucrările [9] şi [14] se indică o utilizare a supraîncărcării operatorului *cast* pentru folosirea întregilor pozitivi de valori mici.

Exemplul de mai jos este inspirat din aceste lucrări.

Exemplu:

Tipul abstract *intmic* se implementează pentru a putea controla lucrul cu întregi situaţi în intervalul [0,63].

```
class intmic {
      char v;   // valoarea poate fi pastrata pe un octet

      void atribuire(int i);
     /* - atribuie lui v valoarea lui i;
        - daca i nu apartine intervalului [0,63], se da mesaj si se
          trunchiaza valoarea lui i la ultimii 6 biti */
  public:
      intmic(int i=0)
      /* - construieste obiectul intmic initializindu-l cu valoarea lui i;
       - realizeaza conversia utilizator implicita din int in intmic */
```

```
    {atribuire(i);}

    operator int()   //supraincarcarea operatorului cast
    /* conversie din tipul intmic in tipul int */
    {
      return v;
    }

    intmic& operator = (int i)
    {atribuire(i); return *this;}
};
```

Funcţia *atribuire* se defineşte ca mai jos:

```
void intmic::atribuire(int i)
{
 if(i < 0 || i > 63)
   printf("valoarea: %d nu apartine\
            intervalului [0,63]\n",i);
 v=i&63;
}
```

Mai jos, se indică exemple de utilizare a tipului definit cu ajutorul clasei *intmic*.

```
intmic m1;          // se instantiaza obiectul m1 cu m1.v=0
intmic m2=12;       // se instantiaza obiectul m2 cu m2.v=12
intmic m3 = 123;    // se da mesaj de eroare
                    // se instantiaza obiectul m3 cu m3.v=59
intmic m4=m1;       // copiere bit_cu_bit
intmic m5=m2-2;     /* - m2 se converteste spre int aplicindu-se
                         supraincarcarea operatorului cast;
                       - apoi diferenţa, se converteste din int
                         in intmic si se copiaza bit_cu_bit */
intmic m6=m2*10;    /* - m2 se converteste spre int;
                       - se inmulteste cu 10 si se obtine 120;
                       - se converteste din int in intmic, se afisea-
                         za mesaj si se atribuie lui m6 valoarea 56
                    */
int i=m2*10;        /* - m2 se converteste spre int;
                       - se inmulteste cu 10 si se obtine 120 care
                         este de tip int;
                       - se atribuie lui i valoarea 120.
                    */
intmic m7;
```

```
m7=m2-2;                /* - m2 se converteste spre int aplicindu-se
                             supraincarcarea operatorului cast;
                           - se face diferenta m2-2 si se obtine o
                             data de tip int(10);
                           - se aplica operatorul = supraincarcat
                             si se atribuie obiectului m7 valoarea
                             diferentei. */
```

Exerciţii:

24.8 În exerciţiul 10.31, s-a definit tipul utilizator *RATIONAL* cu ajutorul declaraţiei:

> **typedef struct** {
> **double** *numarator*;
> **double** *numitor*;
> } *RATIONAL*;

În exerciţiile 10.31, 10.32, 10.33, 10.34, 10.35 şi 10.36 se definesc funcţii care permit operaţii cu date de tip *RATIONAL*.

În exerciţiul de faţă, se defineşte tipul abstract *rational* care permite operaţii cu numere analoge cu cele definite pentru tipul utilizator *RATIONAL*.

Datele de tip *RATIONAL* le numim *numere fracţionare*. Valoarea unei fracţii este egală cu valoarea raportului:

numarator/numitor

În mod analog, obiectele care sînt instanţieri ale clasei *rational* le numim fracţii, iar valoarea fracţiei se defineşte prin acelaşi raport. În acest caz, valoarea unei fracţii se obţine ca rezultat al aplicării supraîncărcării operatorului *cast*. Acesta realizează conversia obiectelor de tip *rational* în valori de tip *double*.

O supraîncărcare analogă a operatorului *cast* se dă şi în cartea [18].

FIŞIERUL BXXIV8.H

```
class rational {
    long numarator;
    long numitor;

    void simplifica();  /* imparte numaratorul si numitorul
                           cu c.m.m.d.c. al lor */
```

```
public:
    rational (long num=0,long numit=1);
   /* constructor utilizat pentru initializarea obiectelor de tip rational
      si pentru conversia utilizator implicita a datelor de tip predefi-
      nit in obiecte de tip rational */

    int citrational(char *s);
    /* - afiseaza sirul spre care pointeaza s daca nu este vid; altfel
         afiseaza texte standard;
       - citeste componentele obiectului rational curent;
       - returneaza:
              0 - la intilnirea sfirsitului de fisier;
              1 - altfel. */

    void afisrational(char *f);
    /* - afiseaza componentele obiectului rational curent;
       - se foloseste formatul spre care pointeaza f sau un format
         standard daca acesta este vid. */

    friend rational operator +(rational r1,
                               rational r2);
    // returneaza r1+r2

    friend rational operator -(rational r1,
                               rational r2);
    // returneaza r1-r2

    friend rational operator *(rational r1,
                               rational r2);
    // returneaza r1*r2

    friend rational operator /(rational r1,
                               rational r2);
    // returneaza r1/r2

    void operator ++();
    // incrementeaza obiectul curent

    void operator --();
    // decrementeaza obiectul curent

    // prin r notam obiectul curent
    rational& operator += (rational r2); //r = r+r2
```

```
        rational& operator -= (rational r2);
        // r = r-r2

        rational& operator *= (rational r2);
        // r = r*r2

        rational& operator /= (rational r2);
        // r = r/r2

        friend rational operator - (rational r1);
        // returneaza -r1

        operator double();
        /* - supraincarca operatorul cast;
           - conversie din rational in double */

        friend int operator == (rational r1,
                                rational r2);
        // returneaza 1 daca r1 == r2, 0 altfel

        friend int operator < (rational r1,
                               rational r2);
        // returneaza 1 daca r1 < r2, 0 altfel

        friend int operator <= (rational r1,
                                rational r2);
        // returneaza 1 daca r1 <= r2, 0 altfel

        long retnumitor();
        // returneaza numitorul obiectului curent

        long retnumarator();
        // returneaza numaratorul obiectului curent;
};
```

Fişierul de extensie CPP conţine definiţiile funcţiilor membru. Totodată se includ fişiere cu funcţii obişnuite care sînt utilizate de către unele dintre funcţiile membru. Acestea sînt:

- *rcit_long* (exerciţiul 24.5);
- *rcit_long_lim* (exerciţiul 24.6);
- *cmmdc* (exerciţiul 10.29);

– *cmmmc* (exercițiul 10.30).

FIȘIERUL BXXIV8

```
#ifndef __RCIT_LONG
#include "BXXIV5.CPP"          // rcit_long
#define __RCIT_LONG
#endif
#ifndef __RCIT_LONG_LIM
#include "BXXIV6.CPP"          // rcit_long_lim
#define __RCIT_LONG_LIM
#endif
#ifndef __CMMDC
#include "BX29.CPP"            // cmmdc
#define __CMMDC
#endif
#ifndef __CMMMC
#include "BX30.CPP"            // cmmmc
#define __CMMMC
#endif
#ifndef __RATIONAL_H
#include "BXXIV8.H"
#define __RATIONAL_H
#endif

void rational::simplifica()
/* imparte numaratorul si numitorul cu c.m.m.d.c. al lor */
{
 long c;

 if(numitor)
    c= cmmdc(numarator,numitor);
 else{
    printf("numitor nul\n");
    return;
 }
 numarator = numarator/c;
 numitor = numitor/c;
}
```

```
rational::rational(long num1,long num2)
/* constructor pentru initializare si conversie din tip predefinit in tipul
   rational */
{
 numarator=num1;
 if(num2)
     numitor = num2;
 else {
     printf("numitor nul; se forteaza la 1\n");
     numitor = 1;
 }
}

int rational::citrational(char *s)
/* - afiseaza sirul spre care pointeaza s daca nu este vid; altfel afiseaza un
     text standard;
   - citeste componentele obiectului rational curent;
   - returneaza:
         0 - la intilnirea sfirsitului de fisier;
         1 - altfel.
*/
{
 long ln;

 if(s && *s) {     // sirul s nu este vid
     for(;;) {       // citeste numaratorul
         printf(s);
         int i;
         if((i=scanf("%ld",&ln)) == EOF)
             return 0;
         if(i == 1)
             break;
         printf("nu s-a citit un intreg\n");
         fflush(stdin);
     }
     numarator = ln;
     for(;;) {   // citeste numitorul
         int i;
         if((i=scanf("%ld",&ln))==EOF)
             return 0;
         if(i==1&&ln!=0)
             break;
         printf("nu s-a tastat un intreg nenul\n");
```

```
        fflush(stdin);
    }
    numitor = ln;
    fflush(stdin);
    return 1;
  } //sfirsit if sirul nu este vid

  //s pointeaza spre sirul vid

  if(rcit_long_lim("numarator=",-2147483648,
                   2147483647, numarator)==0)
    return 0;
  do{
    if(rcit_long_lim("numitor=",-2147483648,
                     2147483647,numitor)==0)
        return 0;
    if(numitor != 0)
        return 1;
    printf("numitor nul\n");
    printf("se reia citirea numitorului\n");
  } while(1);

}     //sfirsit citrational

inline void rational::afisrational(char *f)
/* - afiseaza obiectul rational curent;
   - se foloseste formatul spre care pointeaza f sau un format standard
     daca f pointeaza spre sirul vid. */
{
 if(f && *f)  //formatul nu este vid
    printf(f,numarator,numitor);
 else             //se utilizeaza un format standard
    printf("numarator = %ld\t numitor = %ld\n",
           numarator,numitor);
}

rational operator+(rational r1,rational r2)
//returneaza r1+r2
{
 rational r;    //numarator = 0, numitor = 1

 if(r1.numarator == 0)
    return r2;
```

```
 if(r2.numarator == 0)
    return r1;
 long m = cmmmc(r1.numitor,r2.numitor);
 r.numarator=r1.numarator*(m/r1.numitor)+
             r2.numarator*(m/r2.numitor);
 r.numitor = m;
 r.simplifica();
 return r;
}

rational operator -(rational r1)  // returneaza -r1
{
 rational r;

 r.numarator = -r1.numarator;
 r.numitor = r1.numitor;
 return r;
}

rational operator -(rational r1,rational r2)
// returneaza r1-r2
{
 return r1 + -r2;
}

rational operator *(rational r1,rational r2)
// returneaza r1*r2
{
 rational r;

 r.numarator = r1.numarator*r2.numarator;
 r.numitor = r1.numitor*r2.numitor;
 r.simplifica();
 return r;
}

rational operator /(rational r1,rational r2)
// returneaza r1/r2
{
 rational r;

 r.numarator = r1.numarator*r2.numitor;
 r.numitor = r1.numitor*r2.numarator;
```

```
 if(r.numitor == 0){   //impartire cu zero
     printf("impartire cu zero\n");
     return 0;
 }
 r.simplifica();
 return r;
}

inline void rational::operator++()
//incrementeaza obiectul curent
{
 *this = *this+1;
}

inline void rational::operator--()
//decrementeaza obiectul curent
{
 *this = *this-1;
}

inline rational& rational::operator += (rational
                                        r2)
//r = r+r2
{
 *this = *this+r2;
 return *this;
}

inline rational& rational::operator -= (rational
                                        r2)
//r = r-r2
{
 *this = *this-r2;
 return *this;
}

inline rational& rational::operator *= (rational
                                        r2)
//r = r*r2
{
 *this = *this*r2;
 return *this;
}
```

```
inline rational& rational::operator /= (rational
                                         r2)
// r = r/r2
{
 *this = *this/r2;
 return *this;
}

rational::operator double()
// supraincarcarea operatorului cast
// conversie din rational in double
{
 if(numitor == 0){   // numitor nul
     printf("numitor nul\n");
     return 0.0;
 }
 return (double) numarator/numitor;
}

int operator == (rational r1,rational r2)
// returneaza 1 daca r1 == r2 si zero altfel
{
 if(r1.numarator == 0)
     if(r2.numarator == 0)
         return 1;
     else
         return 0;
 if(r2.numarator == 0)
     if(r1.numarator == 0)
         return 1;
     else
         return 0;

 rational t1 = r1;
 rational t2 = r2;
 t1.simplifica();
 t2.simplifica();
 return t1.numarator==t2.numarator &&
         t1.numitor==t2.numitor;
}
```

```
inline int operator < (rational r1,rational r2)
// returneaza 1 daca r1 < r2, 0 altfel
{
 // se utilizeaza explicit supraincarcarea operatorului cast
 return double(r1) < double(r2);
}

inline int operator <= (rational r1,rational r2)
// returneaza 1 daca r1 <= r2, 0 altfel
{
 return double(r1) <= double(r2);
}

inline long rational::retnumarator()
// returneaza numaratorul obiectului curent
{
 return numarator;
}

inline long rational::retnumitor()
// returneaza numitorul obiectului curent
{
 return numitor;
}
```

Observaţii:

1. Supraîncărcarea operatorului == se poate realiza la fel ca şi supraîncărcarea operatorilor < şi <=, adică utilizînd supraîncărcarea operatorului *cast*:

   ```
   return double(r1) == double(r2);
   ```

   Metoda utilizată mai sus este mai exactă, deoarece împărţirea numărătorului la numitorul fracţiei conduce adesea la un rezultat aproximativ (15 zecimale exacte).

2. O expresie de forma:

   ```
   r+d
   ```

   unde *r* este un obiect de tip raţional şi *d* un număr de un tip predefinit se evaluează astfel:

   - deoarece operatorul + este supraîncărcat pentru operanzi de tip raţional, termenul *d* se converteşte implicit spre un obiect raţional care are numitorul egal cu 1;

- apoi se adună cele două obiecte raţionale (*r* şi cel obţinut prin conversia *implicită*) apelîndu-se funcţia care supraîncarcă operatorul +.

3. Expresia:

```
double(r)+d
```

se evaluează astfel:

- obiectul raţional se converteşte spre *double*;
- apoi se aplică operatorul +, folosind regula conversiilor implicite.

4.

```
x = r+d
```

unde:

- *x* şi *d* sînt variabile de tip predefinit;

- *r* este un obiect de tip rational.

Expresia:

```
r+d
```

se evaluează ca la punctul 2 de mai sus. Apoi, se aplică în mod implicit supraîncărcarea operatorului *cast* care converteşte în *double* obiectul raţional care este rezultatul evaluării expresiei r+d. Rezultatul de tip *double* se atribuie lui *x* direct dacă acesta este de tip *double* sau făcînd o nouă conversie din tipul *double* spre tipul lui *x*.

24.9 În clasele primare se dau elevilor spre rezolvare exerciţii care comportă evaluări de expresii ce conţin ca operanzi fracţii şi cele 4 operaţii aritmetice (adunare, scădere, înmulţire şi împărţire).

Expresia de mai jos este un astfel de exerciţiu. La scrierea ei se utilizează notaţiile obişnuite din limbajul C, adică pentru înmulţire se utilizează semnul "*", iar pentru împărţire semnul "/". De asemenea, fracţia se scrie sub forma liniară şi se include între paranteze rotunde (numărător/numitor):

(((7/15)+(14/15)+(2/9))*(31/3)-(12/11)*((8/3)-(7/4)))/
(((3/7)-(1/4))/(3/28)-1)

Notînd fracţiile cu f1,f2, ... expresia de mai sus se transcrie astfel:

((f1+f2+f3)*f4-f5*(f6-f7))/((f8-f9)/f10-1)

unde:

f1=(7/15), f2=(14/15), f3=(2/9), f4=(31/3), f5=(12/11),

f6=(8/3), f7=(7/4), f8=(3/7), f9=(1/4), f10=(3/28).

Programul de față evaluează expresia de mai sus şi afişează rezultatul.

PROGRAMUL BXXIV9

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#include "BXXIV8.CPP"

main()
/* - evalueaza o expresie cu operanzi fractii;
   - afiseaza rezultatul evaluarii expresiei.
*/
{
 rational f1(7,15),f2(14,15),f3(2,9),f4(31,3),
          f5(12,11),f6(8,3),f7(7,4),f8(3,7),
          f9(1,4),f10(3,28);
 rational f;

 f=((f1+f2+f3)*f4-f5*(f6-f7))/((f8-f9)/f10-1);
 f.afisrational("");
}
```

Observaţie:

Valoarea expresiei este egală cu 1064/45.

24.10 Să se scrie un program care rezolvă ecuaţii de gradul 1 cu coeficienţi fracţionari.

PROGRAMUL BXXIV10

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
```

```
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#include "BXXIV8.CPP"

main()
/* rezolva ecuatii de forma
      a*x+b = 0
   unde a si b sint obiecte de tip rational
*/
{
 rational a,b,x;

 for(;;) {
     if(a.citrational("") == 0)
        exit(0);  //EOF
     if(b.citrational("") == 0)
        exit(1);  //EOF
     if(a == 0 && b == 0){
        printf("ecuatie nedeterminata\n");
        continue;
     }
     if(a == 0){
        printf("ecuatia nu are solutie\n");
        continue;
     }
     x = -b/a;
     x.afisrational("x=(%ld)/(%ld)\n");
      // valoarea fractiei
     printf("x=%g\n",(double)x);
     printf("Actionati o tasta pentru a\
            continua\n");
     getch();
  } // sfirsit for
 }
```

24.11 Să se scrie un program care rezolvă ecuaţii de gradul 2 cu coeficienţi fracţionari.

Deoarece rădăcina pătrată din expresia:

b*b-4*a*c

în general este un număr real, calculele care urmează se vor face în flotantă dublă precizie.

PROGRAMUL BXXIV11

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#include <math.h>
#include "bxxiv8.cpp"

main() /* rezolva ecuatii de gradul 2 cu coeficienti fractionari */
{
 rational a,b,c;

 for(;;){
    printf("coeficientul lui x patrat\n");
    if(a.citrational("") == 0)
       exit(0); //EOF
    printf("coeficientul lui x\n");
    if(b.citrational("") == 0)
       exit(1); //EOF
    printf("termenul liber\n");
    if(c.citrational("") == 0)
       exit(1); //EOF
    if(a==0 && b==0 && c==0){
       printf("ecuatie nedeterminata\n");
       continue;
    }
    if(a==0 && b==0){
       printf("ecuatia nu are solutie\n");
       continue;
    }
    if(a==0){  //ecuatie de grad 1
       rational x = -c/b;
```

```
        printf("ecuatie de gradul 1\n");
        x.afisrational("x=(%ld)/(%ld)\n");
        printf("Actionati o tasta pentru a\
                continua\n");
        getch();
        continue;
    }

    // a != 0; ecuatie de gradul 2
    double delta = b*b-4*a*c;
    double doia = 2*a;
    double minusb = -b;
    if(delta > 0) {  // radacini reale
        delta = sqrt(delta);
        double x = (minusb + delta)/doia;
        printf("x1 = %g\t",x);
        x=(minusb - delta)/doia;
        printf("x2 = %g\n",x);
    }
    else
        if(delta==0){  // radacina dubla
            double x = minusb/doia;
            printf("radacina dubla\n");
            printf("x=%g\n",x);
    }
        else{   // radacini complexe
            delta = sqrt(-delta);
            double real = minusb/doia;
            double imag = delta/doia;
            printf("x1=%g+i*(%g)\n",real,imag);
            printf("x2=%g-i*(%g)\n",real,imag);
    }
    printf("Actionati o tasta pentru a\
            continua\n");
    getch();
  }
}
```

Observaţie:

- Declaraţia:

double delta=b*b-4*a*c;

evaluează expresia:

b*b-4*a*c

apelînd funcţiile care supraîncarcă operaţiile pe care le conţine, pentru tipul rational, apoi se apelează funcţia care supraîncarcă operatorul *cast* pentru a converti valoarea expresiei din *rational* în double.

24.3. Conversia dintr-un tip abstract într-un alt tip abstract

Pentru conversia obiectelor dintr-un tip abstract într-un alt tip abstract se pot folosi atît conversiile realizate cu ajutorul constructorilor, cît şi cele realizate prin supraîncărcarea operatorului *cast*.

De exemplu, fie două tipuri abstracte *tip1* şi *tip2*. Pentru a realiza conversia obiectelor de tip *tip1* în obiecte de tip *tip2* folosind constructorul clasei *tip2*, se procedează la definirea unui constructor pentru clasa *tip2* care să aibă ca parametru un obiect de tip *tip1*. Deci, clasa *tip2* are un constructor de antet:

tip2(tip1 ob)

Acest constructor trebuie să aibă acces la datele membru ale clasei *tip1*, date care, de obicei, sînt protejate. De aceea, constructorul respectiv se declară ca funcţie prieten a clasei *tip1*.

Exemplu:

Alături de tipul abstract *grad*, definit în exerciţiul 24.4, considerăm tipul abstract *radian* pentru lucrul cu măsuri de unghiuri exprimate în radiani.

Clasa *radian* se defineşte în aşa fel încît să se poată converti cu ajutorul constructorului măsurile unghiulare din grade sexagesimale în radiani. În acest scop putem proceda ca mai jos:

```
class radian {
    double rad;
 public:
    radian(double d = 0)
    // conversie din tip predefinit in tipul radian
    {
       rad = d;
    }
```

```
radian(grad);
// constructor care realizeaza conversie din tipul grad in tipul
// radian
...
};
```

Constructorul:

```
radian(grad)
```

este o funcţie prieten pentru clasa *grad*, deoarece aceasta trebuie să aibă acces la data membru *totsec* a clasei *grad* (vezi exerciţiul 24.4.). Pentru a realiza acest lucru, în definiţia clasei *grad* se adaugă declaraţia:

```
friend radian::radian(grad);
```

Constructorul care defineşte conversia din tipul *grad* în tipul *radian* se defineşte simplu, ca mai jos:

```
radian::radian(grad g)
// conversie din grade sexagesimale in radiani
{
 rad=g.totsec*PI/648000;
}
```

unde PI are valoarea 3.14159265358979

Conversia din grade sexagesimale în radiani se poate defini şi cu ajutorul supraîncărcării operatorului *cast*.

Pentru a realiza conversia obiectelor din tipul abstract *tip1* în tipul abstract *tip2*, cu ajutorul unei funcţii membru nestatică care supraîncarcă operatorul *cast*, se foloseşte antetul:

```
tip1::operator tip2()
```

Această funcţie este o funcţie membru nestatică a clasei *tip1* şi o funcţie prieten pentru clasa *tip2*.

Pentru a realiza conversia din grade sexagesimale în radiani, se procedează ca mai jos:

- se defineşte funcţia membru nestatică a clasei *grad*, de antet:

  ```
  grad::operator radian()
  ```

- iar în clasa *radian* se declară funcţia respectivă ca funcţie prieten.

  ```
  class radian;
  class grad {
  ```

```
            long totsec;
            long sgrad;
            int smin;
            int ssec;
            void grminsec();
        public:
            grad(long sec=0) {
                totsec = sec;
                grminsec();
            }
            grad(long g,int m,int s)
            {
                totsec=g*3600+m*60+s;
                grminsec();
            }
            ...
            operator radian();
            ...
    };
    class radian {
            double rad;
        public:
            radian(double d=0)
            {
                rad = d;
            }
            ...
            friend grad::operator radian();
            ...
    };
```

În acest caz, funcţia care supraîncarcă operatorul *cast* se defineşte astfel:

```
grad::operator radian()
{
 radian r = totsec*PI/648000;
 return r;
}
```

Declaraţia:

radian r = totsec*PI/648000;

se tratează astfel:

- **data membru *totsec* este componentă a obiectului curent, deci i se**

aplică pointerul *this* în mod implicit:

```
this -> totsec;
```

- se evaluează expresia din dreapta semnului de atribuire şi se obţine o valoare de tip *double* care reprezintă măsura în radiani a unghiului a cărui măsură în secunde sexagesimale este egală cu valoarea lui

```
this-> totsec.
```

Pentru a atribui această valoare obiectului *r* de tip *radian*, se realizează conversia utilizator implicită care converteşte o dată din tip *double* într-un obiect de tip *radian* Aceasta se realizează apelîndu-se în mod implicit constructorul clasei *radian* care are un parametru de tip *double*. Obiectul obţinut în acest fel se atribuie lui *r* printr-o copiere bit_cu_bit.

Mai sus, s-a definit conversia din tipul *grad* în tipul *radian* prin două metode:

- una foloseşte constructorul clasei *radian* care are un parametru de tip *grad* şi acesta este o funcţie prieten pentru clasa *grad*;
- cealaltă foloseşte o funcţie membru nestatică, care supraîncarcă operatorul *cast* pentru clasa *grad* şi care este o funcţie prieten pentru clasa *radian*.

Utilizatorul va trebui să opteze pentru una din metode, deoarece prezenţa ambelor metode conduce la ambiguitate şi în acest caz, compilatorul semnalează eroare în loc de a aplica una din metodele de conversie.

Uneori este nevoie să se aplice mai multe conversii implicite pentru a evalua un operand.

Menţionăm că, în acest caz, compilatorul admite o singură conversie implicită definită de utilizator. Prin conversie definită de utilizator înţelegem una din următoarele conversii:

- din tip predefinit în tip abstract;
- din tip abstract în tip predefinit;
- din tip abstract în tip abstract.

Exemplu:

Presupunem că,clasa *radian* are o funcţie membru care supraîncarcă operatorul *cast* pentru a converti obiectele de tip *radian* în date de tip double. O astfel de funcţie se poate defini astfel:

```
radian::operator double()
{
 return rad;
```

```
}
```

Fie instanţierea:

```
radian r;
...
```

Apelul:

```
sin(r)
```

este corect şi se realizează după conversia prealabilă a obiectului *r*, de tip *radian*, în tipul *double* (funcţia *sin* are prototipul double sin(double)).

Fie instanţierea:

```
grad g;
...
```

Apelul:

```
sin(g)
```

este eronat, deoarece comportă mai mult de o conversie implicită definită de utilizator:

- conversia lui *g* din *grad* în *radian*;
- conversia din *radian* în *double*.

În schimb apelul:

```
sin(radian(g))
```

este corect, deoarece aici se aplică o singură conversie utilizator *implicită*:

- conversia obiectului radian(g) de tip *radian* într-o dată de tip *double*.

Observaţie:

La supraîncărcarea operatorului *cast* cu ajutorul funcţiei membru:

```
operator radian()
```

a clasei *grad* nu este necesar ca funcţia să fie declarată ca funcţie prieten a clasei *radian*. Aceasta deoarece la realizarea conversiei în radiani nu este necesar să se facă acces la data membru protejată a clasei *radian*. În general este nevoie de acest lucru şi atunci funcţia care supraîncarcă operatorul *cast* va fi o funcţie prieten a clasei spre care se face conversia.

Exerciţii:

24.12 Să se implementeze tipurile abstracte *celsius* şi *fahrenheit* care permit utilizarea datelor de tip *celsius* şi respectiv *fahrenheit* la exprimarea

măsurii temperaturii şi la efectuarea de calcule. De asemenea, se definesc conversii în ambele sensuri: din grade Celsius în grade Fahrenheit şi invers.

O formulă simplă de conversie din grade Fahrenheit în grade Celsius este următoarea:

c = (5/9)*(f-32)

unde:

f - Reprezintă măsura temperaturii în grade Fahrenheit.
c - Reprezintă măsura temperaturii în grade Celsius.

Relaţia care asigură conversia inversă este următoarea:

f = (9/5)*c+32 sau 1.8*c+32

Clasa care implementează tipul *fahrenheit* conţine funcţii pentru supraîncărcarea operatorilor aritmetici după cum urmează:

- adunarea obiectelor de tip fahrenheit;
- scăderea obiectelor de tip fahrenheit;
- înmulţirea unui obiect de tip fahrenheit cu o dată numerică de tip predefinit;
- împărţirea unui obiect de tip fahrenheit cu o dată numerică de tip predefinit;
- negativarea unui obiect de tip fahrenheit.

FIŞIERUL BXXIV12.H

```
class celsius;
class fahrenheit {
    double fah;
  public:
    fahrenheit(double f=0);
    /* constructor utilizat pentru initializare si conversie din tip
       predefinit in tipul fahrenheit */

    fahrenheit(celsius);
    /* constructor utilizat pentru conversia din tipul celsius in tipul
       fahrenheit */

    double retfahrenheit();
    /* returneaza componenta fah a obiectului curent */
```

```
    operator celsius();
    /* conversie din tipul fahrenheit in tipul celsius */

    fahrenheit operator -();  // negativare

    friend fahrenheit operator +(fahrenheit f1,
                                 fahrenheit f2);
    // returneaza f1+f2

    friend fahrenheit operator  -(fahrenheit f1,
                                  fahrenheit f2);
    // returneaza f1-f2

    fahrenheit operator *(double d);
    // returneaza f*d, unde f este obiectul curent

    friend fahrenheit operator *(double d,
                                 fahrenheit f2);
    // returneaza d*f2

    fahrenheit operator /(double d);
    // returneaza f/d, unde f este obiectul curent
};

class celsius {
    double cel;
 public:
    celsius(double d=0);
    /* constructor utilizat pentru initializare si conversie din
       double in celsius */

    friend fahrenheit::fahrenheit(celsius);
    /* constructorul clasei fahrenheit care face conversii din celsius in
       fahrenheit trebuie sa aiba acces la data membru protejata cel */

    double retcelsius();
    /* returneaza componenta cel a obiectului curent */
};
```

Mai jos, se definesc funcţiile membru şi prieten ale claselor *fahrenheit* şi *celsius*.

FIŞIERUL BXXIV12

```
#ifndef __FAHCEL_H
#include "BXXIV12.H"
#define __FAHCEL_H
#endif

inline fahrenheit::fahrenheit (double d)
/* se utilizeaza la initializarea obiectelor de tip fahrenheit
   si la conversia din double in fahrenheit */
{
 fah = d;
}

inline fahrenheit::fahrenheit(celsius c)
/* se utilizeaza la conversia din celsius in fahrenheit */
{
 fah = c.cel*1.8+32;
}

inline double fahrenheit::retfahrenheit()
/* - returneaza fah pentru obiectul curent;
   - se utilizeaza in locul conversiei din fahrenheit in double */
{
 return fah;
}

fahrenheit::operator celsius()
// conversie din fahrenheit in celsius
{
 celsius c=(5.0/9.0)*(fah-32);
 return c;
}

fahrenheit fahrenheit::operator -() // negativare
{
 fahrenheit f = *this;
 f.fah = -f.fah;
 return f;
}
```

```
fahrenheit operator +(fahrenheit f1,fahrenheit f2)
// returneaza f1+f2
{
 fahrenheit f;

 f.fah = f1.fah +f2.fah;
 return f;
}

fahrenheit operator -(fahrenheit f1,fahrenheit f2)
// returneaza f1-f2
{
 fahrenheit f = f1 + -f2;
 return f;
}

fahrenheit fahrenheit::operator *(double d)
// returneaza obiectul curent inmultit cu d
{
 fahrenheit f;

 f.fah = fah*d;
 return f;
}

fahrenheit operator *(double d,fahrenheit f2)
// returneaza d*f2
{
 fahrenheit f=f2*d;
 return f;
}

fahrenheit fahrenheit::operator /(double d)
// returneaza obiectul curent impartit la d
{
 fahrenheit f;

 f.fah = fah/d;
 return f;
}
```

```
celsius::celsius(double c)
/* constructor utilizat la initializare si la conversia din
   double in celsius */
{
 cel = c;
}

inline double celsius::retcelsius()
/* returneaza cel pentru obiectul curent */
{
 return cel;
}
```

Observaţii:

1. Expresii de forma:

   f1*f2, f1/f2

   unde:

   f1 şi *f2* - sînt obiecte de tip *fahrenheit*, nu sînt admise.

2. Dacă se supraîncarcă operatorul *cast* aşa încît să permită conversia obiectelor de tip fahrenheit în date de tip *double*, atunci expresia f1/f2 este admisă şi pentru cazul în care *f1* şi *f2* sînt obiecte de tip *fahrenheit*. De asemenea, supraîncărcarea operatorului *cast* în acest fel face ca expresia:

   f1*f2

   să fie ambiguă pentru cazul cînd *f1* şi *f2* sînt de tip *fahrenheit*.

   Ambiguitatea rezultă din faptul că operaţia de înmulţire este supraîncărcată cu 2 funcţii care au prototipurile:

   ```
   fahrenheit operator*(double d)
   ```

   şi

   ```
   fahrenheit operator*(double d,fahrenheit f2)
   ```

   Prima, se aplică pentru o expresie de forma:

   f*d

   iar cea de a doua, pentru o expresie de forma:

   d*f .

   unde:

f - este de tipul *fahrenheit*;

d - este de tip *double*.

În condiţiile în care operatorul *cast* este supraîncărcat pentru a realiza conversia din fahrenheit în *double*, ambele funcţii care supraîncarcă operatorul se pot aplica la evaluarea expresiei f1*f2:

- se converteşte *f2* în *double* şi se aplică prima funcţie care supraîncarcă operatorul *;

sau

- se converteşte *f1* în *double* şi se aplică cea de a doua funcţie care supraîncarcă operatorul *.

Din cauza ambiguităţii, compilatorul semnalează o eroare la întîlnirea unei expresii de forma f1*f2.

3. Se pot utiliza expresii de forma:

c1+c2, c1-c2, c1*d, d*c1, c1/d

unde:

c1, c2 - sînt obiecte de tip *celsius*;

d - este o dată de tip *double*.

Operanzii de tip *celsius* se convertesc în obiecte de tip *fahrenheit* şi apoi se realizează evaluarea expresiei folosind supraîncărcările operatorilor respectivi pentru obiecte *fahrenheit*.

4. Pentru a interzice utilizarea expresiilor de forma f1*f2 sau f1/f2,

unde:

f1 şi *f2* sînt obiecte de tip *fahrenheit*,

nu s-a supraîncărcat operatorul *cast* pentru conversia obiectelor de tip *fahrenheit* în date de tip *double*.

Pentru a avea acces la componenta protejată *fah*, în lipsa supraîncărcării operatorului *cast*, s-a definit funcţia membru *retfahrenheit*.

24.13 Să se scrie un program care afişează o tabelă pentru conversia gradelor *Celsius* în grade *Fahrenheit*.

Tabelarea se face din grad în grad, limitele intervalului în care se face tabelarea sînt 2 întregi de tip *int* care se citesc de la intrarea standard.

PROGRAMUL BXXIV13

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "BXX4.CPP"      //rcit_int
#include "BXX5.CPP"      //rcit_int_lim
#include "BXXIV12.CPP"   //FAHCEL

main()
/* - citeste pe m si n de tip int;
   - afiseaza o tabela de conversie a gradelor Celsius in grade
     Fahrenheit;
   - gradele Celsius variaza, cu pasul 1, in intervalul [m,n].
*/
{
 int m,n;

 //citeste limitele m si n
 //limita inferioara

 if(rcit_int_lim("limita inferioara m = ",
                  -32768,32767,m)==0)
     exit(0);
 if(rcit_int_lim("limita superioara n = ",
                   m,32767,n)==0){
    printf("s-a tasta EOF\n");
    exit(1);
 }

 // Afiseaza antet
 printf("\n\n\t Conversie Celsius-Fahrenheit\n");
 printf("\n Grade Celsius\t Grade Fahrenheit\n");
 for(int i=m;i <= n;i++){
    celsius c(i);
    fahrenheit f=c;  //conversie din Celsius in Fahrenheit
    printf("%13d\t %16g\n",i,f.retfahrenheit());
    if((i-m+1)%23 == 0) {
        printf("Actionati o tasta pentru a\
                continua\n");
        getch();
    }
  }
}
```

24.14 Să se scrie un program care citeşte un şir de întregi ce reprezintă măsurile temperaturii în grade Celsius, afişează măsurile respective în grade Fahrenheit, precum şi media lor în grade Celsius şi Fahrenheit. Şirul de întregi se termină cu sfirşitul de fişier.

PROGRAMUL BXXIV14

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "BXXIV5.CPP"  // rcit_long
#include "BXXIV6.CPP"  // rcit_long_lim
#include "BXXIV12.CPP" // FAHCEL

main()
/* - citeste un sir de intregi ce reprezinta masuri ale temperaturii in grade
    Celsius;
   - afiseaza masurile respective in grade Fahrenheit, precum si media
     lor in grade Celsius si Fahrenheit
*/
{
 int n=0;
 long t;
 fahrenheit sum;  // sum.fah = 0

 for(;;) {
   if(rcit_long_lim("temperatura in grade\
                      Celsius =",
             -2147483648,2147483647,t)==0)
       break;  // EOF
   celsius c(t);
   sum = sum + c;
   fahrenheit f=c; // conversie Celsius in Fahrenheit
   printf("temperatura in grade Fahrenheit =\
          %g\n", f.retfahrenheit());
   n++;
 }
 if(n){ // S-au citit temperaturi
   sum=sum/n; // calculeaza media
   celsius cmedia=sum;    // conversia mediei in grade celsius
   printf("\nMedia in grade Celsius = %g\n",
          cmedia.retcelsius());
   printf("\nMedia in grade Fahrenheit = %g\n",
```

```
        sum.retfahrenheit());
  }
  else
      printf("Nu s-a citit nimic\n");
}
```

24.15 Să se definească două tipuri abstracte *dc* şi *dczi* pentru reprezentarea datelor calendaristice. Tipul abstract *dc* are trei date membru care definesc ziua, luna şi anul calendaristic. O dată calendaristică validă este în intervalul:

1 ianuarie 1600 - 31 decembrie 4900

Tipul abstract *dczi* are o singură dată membru care defineşte numărul de zile socotite începînd cu 1 ianuarie 1600.

Data de 1 ianuarie 1600 corespunde cu valoarea 1 pentru data membru a tipului *dczi* (originea datelor calendaristice este 31 decembrie 1599).

Tipul abstract *dczi* are funcţii membru care supraîncarcă operaţiile de adunare, scădere precum şi operaţiile de comparaţie şi egalitate. Se defineşte conversia din tipul *dczi* în tipul *dc* şi invers.

Pentru a evita suma a două obiecte de tip *dczi* nu se defineşte funcţia care supraîncarcă operatorul *cast* pentru conversia din *dczi* într-un tip predefinit. În locul ei, se defineşte o funcţie membru care returnează numărul de zile care corespunde unui obiect de tip *dczi*.

Nu se pot aduna două obiecte de tip *dczi*, ci numai o dată predefinită se poate aduna cu un obiect de tip *dczi*. Data de tip predefinit se consideră că reprezintă un număr de zile care se adună la data calendaristică pe care o defineşte operandul *dczi*.

Diferenţa a două obiecte de tip *dczi* reprezintă numărul de zile dintre cele două date calendaristice definite de cei doi operanzi de tip *dczi*. Se poate scădea dintr-un obiect de tip *dczi* o dată de tip predefinit care reprezintă un număr de zile.

Se pot compara numai obiecte de tip *dczi*.

Tipul *dc* are funcţii membru pentru citirea şi afişarea obiectelor de tip *dc*. Se poate face acces la datele membru ale unui obiect de tip *dc* cu ajutorul unor funcţii membru.

FIŞIERUL BXXIV15.H

```
class dc;
class dczi {
    long totzile;
```

```
public:
    dczi();
    /* constructor implicit pentru initializarea obiectelor cu
       totzile = 0 */

    dczi(dc);
    /* constructor pentru conversie din dc in dczi */

    long rettotzile();
    /* returneaza totzile pentru obiectul curent */

    // cu z se noteaza obiectul curent

    dczi operator +(long n);  // returneaza z+n

    friend dczi operator +(long n,dczi z2);
    // returneaza n+z2

    long operator -(dczi z2);  // returneaza z-z2

    dczi operator -(long n);
    // returneaza z-n, unde z este obiectul curent

    int operator == (dczi z2);
    // returneaza 1 dacă z == z2 si zero altfel

    int operator < (dczi z2);
    // returneaza 1 daca z < z2 si zero altfel

    int operator <= (dczi z2);
    // returneaza 1 daca z <= z2 si zero altfel
};

class dc {
    int zi;
    int luna;
    int an;
  public:
    dc(int z=0,int l=0,int a=0);
    /* constructor pentru initializarea obiectelor de tip dc */

    dc(dczi);
    // constructor pentru conversia din dczi in dc
```

```
    int retzi();
    // returneaza zi pentru obiectul curent

    int retluna();
    // returneaza luna pentru obiectul curent

    int retan();
    // returneaza an pentru obiectul curent

    int citdc(char *text);
    // citeste o data calendaristica dupa afisarea
    // sirului spre care pointeaza text

    void afisdc(char *f);
    /* - afiseaza data calendaristica pentru obiectul curent;
       - se foloseste formatul f daca acesta nu pointeaza spre sirul vid;
         altfel se utilizeaza un format standard. */

    friend dczi::dczi(dc);
    // constructor pentru conversii din tipul dc in tipul dczi
};
```

Funcţiile membru se definesc în fişierul de mai jos, de extensie *cpp*. Tot în acest fişier se definesc două funcţii obişnuite care se apelează pentru a verifica datele calendaristice definite cu ajutorul tipurilor *dczi* şi *dc*.

Intervalul 1 ianuarie 1600 - 31 decembrie 4900 conţine 1205666 zile incluzînd şi zilele limită ale intervalului.

Într-adevăr, intervalul respectiv conţine:

4901-1600=3301 ani;

Dacă toţi anii ar avea 365 de zile, atunci numărul de zile ar fi:

3301*365=1204865 zile

Pentru a afla anii multiplii de 4, calculăm cîtul:

3301/4+1=826

Se adună 1 deoarece anul 1600, de început, este multiplu de 4.

Dacă toţi anii multipli de 4 ar fi ani bisecţi, ar însemna că la numărul de zile determinat mai sus s-ar adăuga 826 de zile.

Există 3301/100+1=34 ani care sînt multipli de 100. Se adună 1 deoarece anul 1600 este multiplu de 100. Dintre aceştia numai

3301/400+1=9 sînt bisecţi (1600, 2000, 2400, 2800, 3200, 3600, 4000, 4400 şi 4800). Rezultă că 34-9=25 ani multipli de o sută nu sînt ani bisecţi din cei multipli de 4. Deci se obţin:

826-25=801 zile

care se adaugă din cauza anilor bisecţi.

În total, în intervalul respectiv sînt 1204865+801=1205666 zile.

FIŞIERUL BXXIV15

```
#define MAXZILE 1205666L
#ifndef __DCDCZI_H
#include "BXXIV15.H"
#define __DCDCZI_H
#endif

static int tnrz[] =
      {0,31,28,31,30,31,30,31,31,30,31,30,31};

static int v_calendzile(long tz)
/* - returneaza 1 daca tz nu depaseste totalul de zile din intervalul 1
      ianuarie 1600 - 31 decembrie 4900;
   - in caz contrar, se afiseaza un mesaj de eroare si returneaza valoarea
      zero. */
{
 if(tz <= MAXZILE && tz >= 0)
     return 1;
 printf("Prea multe zile sau numar negativ de\
         zile=%ld\n",tz);
 return 0;
}

static int v_calend(int z,int l,int a)
/* - returneaza 1 daca data calendaristica definita de parametri se afla in
     intervalul 1 ianuarie 1600 - 31 decembrie 4900;
   - in caz contrar, afiseaza un mesaj de eroare si returneaza valoarea
     zero */
{
 if(a==0 && l==0 && z==0)
     return 0;

 if(a < 1600 || a > 4900){
     printf("anul = %d eronat\n",a);
     return 0;
```

```
 }
 if(l < 1 || l > 12){
     printf("luna=%d eronata\n",l);
     return 0;
 }
 if(z < 1 || z > tnrz[l] + (l==2 && (a%4 == 0 &&
   a%100 || a%400 == 0))){
     printf("ziua=%d eronata\n",z);
     return 0;
 }
 // data calendaristica valida
 return 1;
}

inline dczi::dczi()
/* constructor implicit utilizat la initializarea obiectelor cu zero */
{
 totzile = 0;
}

dczi::dczi(dc d)
/* constructor pentru conversia din dc in dczi */
{
 int ancrt;
 int lunacrt;

 totzile = 0;
 if(v_calend(d.zi,d.luna,d.an)){
     for(ancrt=1600;ancrt < d.an;ancrt++)
         totzile += 365+(ancrt%4 == 0 &&
                         ancrt%100 || ancrt%400 == 0);
     for(lunacrt=1;lunacrt < d.luna;lunacrt++)
         totzile += tnrz[lunacrt]+(lunacrt == 2 &&
                         (ancrt%4==0 && ancrt%100 ||
                          ancrt%400==0));
     totzile += d.zi;
 }
}

inline long dczi::rettotzile()
// returneaza totzile pentru obiectul curent
{
 return totzile;
```

```
}
dczi dczi::operator +(long n)  // returneaza z+n
{
 dczi r;

 r.totzile = totzile+n;
 if(v_calendzile(r.totzile) == 0)
     r.totzile = 0;
 return r;
}

dczi operator +(long n,dczi z2)  // returneaza n+z2
{
 return z2+n;
}

inline long dczi::operator -(dczi z2)  // returneaza z-z2
{
 return totzile -z2.totzile;
}

inline dczi dczi::operator -(long n)  // returneaza z-n
{
 return *this + -n;
}
inline int dczi::operator == (dczi z2)
// returneaza 1 daca z == z2 si zero altfel
{
 return totzile == z2.totzile;
}

inline int dczi::operator < (dczi z2)
// returneaza 1 daca z < z2 si zero altfel
{
 return totzile < z2.totzile;
}

inline int dczi::operator <= (dczi z2)
// returneaza 1 daca z <= z2 si zero altfel
{
 return totzile <= z2.totzile;
}
```

```
dc::dc(int z,int l,int a)
// constructor pentru initializarea obiectelor de tip dc
{
 if(v_calend(z,l,a) == 0){
     z = 0;
     l = 0;
     a = 0;
 }
 zi = z;
 luna = l;
 an = a;
}

dc::dc(dczi tz)   // constructor pentru conversie din dczi in dc
{
 zi = luna = an = 0;
 long t = tz.rettotzile();
 if(v_calendzile(t)){
     for(an=1600;t > 365+(an%4==0 && an%100 ||
          an%400==0);an++)
          t -= (365+(an%4==0 && an%100 ||
                     an%400==0));
     for(luna=1;t > tnrz[luna]+(luna==2 &&
        (an%4==0 && an%100 || an%400==0));luna++)
          t -= (tnrz[luna]+(luna==2 &&
               (an%4==0 && an%100 || an%400==0)));
     zi = t;
 }
}

inline int dc::retzi()
// returneaza ziua obiectului curent
{
 return zi;
}

inline int dc::retluna()
// returneaza luna obiectului curent
{
 return luna;
}
```

```
inline int dc::retan()
// returneaza anul obiectului curent
{
 return an;
}

dc::citdc(char *text)
/*- citeste o data calendaristica dupa afisarea sirului spre care pointeaza
   text, daca acesta nu este vid;
  - returneaza:
        0 - la intilnirea sfirsitului de fisier;
        1 - altfel. */
{
 for(;;){
    if(text && *text)
        printf(text);
    if(rcit_int_lim("ziua=",1,31,zi)==0)
        return 0;   // EOF
    if(rcit_int_lim("luna=",1,12,luna)==0)
        return 0;   // EOF
    if(rcit_int_lim("an=",1600,4900,an)==0)
        return 0;   // EOF
    if(v_calend(zi,luna,an))
        break;      // data calendaristica valida
        printf("data calendaristica eronata\n");
        printf("se reia citirea datei\
                calendaristice\n");
    }
    return 1;
}

void dc::afisdc(char *f)
/* - afiseaza data calendaristica pentru obiectul curent;
   - se utilizeaza formatul spre care pointeaza f daca acesta nu este vid,
     in caz contrar, un format standard. */
{
 if(f && *f)
    printf(f,zi,luna,an);
 else
    printf("ziua=%d\t luna=%d\t anul=%d\n",
            zi,luna,an);
}
```

24.16 Să se scrie un program care citeşte date calendaristice şi afişează data calendaristică a zilei următoare pentru fiecare dată citită.

PROGRAMUL BXXIV16

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "BXX4.CPP"         // rcit_int
#include "BXX5.CPP"         // rcit_int_lim
#include "BXXIV15.CPP"      // dczi_dc

main()
/* citeste date calendaristice si afiseaza data calendaristica a zilei
   urmatoare pentru fiecare data citita */
{
 dc d_crt;

 for(;;){
     if(d_crt.citdc(0) == 0)
        exit(0);  //EOF
     dczi d_urm_zile = dczi(d_crt)+1;
     dc d_urm = d_urm_zile;
     printf("ziua urmatoare\n");
     d_urm.afisdc(0);
     printf("Actionati o tasta pentru a\
             continua\n");
     getch();
 }
}
```

Observaţie:

Expresia:

```
d_crt+1
```

nu este admisă de compilator. Aceasta datorită modului în care a fost supraîncărcat operatorul "+". Într-adevăr, operatorul + a fost supraîncărcat pentru obiecte de tip *dczi* folosind două funcţii de prototip:

```
dczi dczi::operator+(long n);
```

şi

```
friend dczi operator+(long n,dczi z2);
```

Prima funcţie este o funcţie membru nestatică şi se poate aplica numai

dacă obiectul din stînga operatorului + este de tip *dczi*. Ori *d_crt* este un obiect de tip *dc*. Deşi există un constructor al clasei *dczi* care converteşte obiecte de tip *dc* în obiecte de tip *dczi*, acesta nu se aplică implicit în această situaţie şi de aceea este nevoie de aplicarea lui explicită. Cea de a doua funcţie care supraîncarcă operatorul + are un prototip care nu corespunde acestei expresii. Într-adevăr, această funcţie se poate apela dacă primul operand este de tip *long* sau se poate converti spre tipul *long*.

Nu există însă o conversie utilizator din tipul *d_crt* spre tipul *long* sau alt tip predefinit.

Dacă ar exista o astfel de conversie, atunci s-ar putea apela funcţia prieten deoarece conversia lui 1 spre tipul *dczi* se poate realiza implicit aplicîndu-se constructorul corespunzător al clasei *dczi*.

Compilatorul admite expresia:

1+d_crt

care are aceeaşi valoare cu expresia de mai sus. În acest caz, se aplică funcţia prieten indicată mai sus, termenul *d_crt* convertindu-se din *dc* în *dczi*.

24.17 Să se scrie un program care citeşte două date calendaristice şi afişează diferenţa, în număr de zile, dintre cea de a doua dată calendaristică şi prima.

PROGRAMUL BXXIV17

```
#include <stdio.h>
#include <stdlib.h>
#include "BXX4.CPP"      // rcit_int
#include "BXX5.CPP"      // rcit_int_lim
#include "BXXIV15.CPP"  // dczi_dc

main()
/* citeste doua date calendaristice si afiseaza diferenta in numar
   de zile dintre cea de a doua data calendaristica si prima
   data calendaristica */
{
 dc d1;
 dc d2;

 if(d1.citdc(0) == 0){
     printf("s-a intilnit EOF\n");
     exit(1);
 }
```

```
   if(d2.citdc(0) == 0){
      printf("s-a intilnit EOF\n");
      exit(1);
   }
   printf("data a doua-prima=%ld\n",dczi(d2)-d1);
}
```

25. MODIFICATORUL CONST PENTRU OBIECTE ŞI FUNCŢII MEMBRU

În capitolul 20 s-au indicat facilităţile oferite de modificatorul *const* la declararea datelor constante.

În principiu, modificatorul *const* prezent într-o declaraţie, protejează data din declaraţia respectivă deoarece ei nu i se poate schimba valoarea la execuţie în mod "direct". De aceea, ea se comportă la fel ca o constantă.

Modificatorul *const* se poate aplica şi la declararea obiectelor pentru a instanţia obiecte constante sau pointeri spre obiecte constante.

Exemplu:

Fie clasa *sir*, definită ca mai jos:

```
class sir {
     char *psir;
     int lungime;
  public:
     sir(char *p);
     /* constructor pentru initializarea obiectului cu un sir de
        caractere */

     sir();   // constructor implicit
     sir(const sir&);   // constructor de copiere

     sir& operator = (const sir&);
     // supraincarcarea operatorului =

     void afsir(char *f);
     /* afiseaza sirul curent folosind formatul definit de f */
};
```

Parametrul constructorului de copiere este o referinţă la un obiect. Acest obiect constituie obiectul sursă care se copiază în obiectul curent.

Constructorul de copiere nu are voie să modifice obiectul sursă. Acest lucru este asigurat folosind modificatorul *const* la declararea parametrului formal.

Modificatorul *const* are aceeaşi utilizare şi în cazul funcţiei care

supraîncarcă operatorul "=". Într-adevăr, obiectul parametru reprezintă partea dreaptă a operatorului de atribuire. Funcţia respectivă nu are voie să modifice operatorul din dreapta semnului de atribuire şi de aceea, parametrul corespunzător lui se declară folosind modificatorul *const*.

Funcţiile membru ale clasei *sir* se pot defini la fel ca în exerciţiul 23.14.

Fie instanţierile:

```
const sir s1("C++ este un C mai bun");
const sir s2=s1;
sir s3;
```

La prima instanţiere se aplică constructorul care are ca parametru un pointer spre un şir de caractere (char *).

La instanţierea a doua se aplică constructorul de copiere.

La instanţierea a treia se aplică constructorul implicit.

O atribuire de forma:

```
s3=s1;
```

este corectă. La realizarea ei se apelează funcţia care supraîncarcă operatorul de atribuire.

În schimb, la o atribuire de forma:

```
s1=s3;
```

compilatorul afişează un avertisment deoarece *s1* a fost declarat cu ajutorul modificatorului *const*.

Modificatorul *const* se poate utiliza în antetul unei funcţii în două moduri.

O primă utilizare a modificatorului *const* este aceea de a proteja obiectul returnat de funcţie. În acest caz, modificatorul *const* precede tipul din antetul sau prototipul funcţiei respective:

(1) ... const *tip nume_clasa::nume_functie*(...)

Cea de a doua utilizare a modificatorului *const* este aceea cînd acesta se află în antetul sau prototipul funcţiei după paranteza închisă care termină lista parametrilor formali ai funcţiei:

(2) ... *nume_clasa::nume_functie*(...) const

O astfel de funcţie nu poate modifica datele membru ale obiectului pentru care se apelează (obiectul curent).

În exemplul de mai sus, funcţia *afsir* este o funcţie care nu trebuie să

modifice obiectul pentru care se apelează. De aceea, ea se poate defini utilizînd modificatorul *const*:

```
void afsir(char *f) const;
```

O funcţie care are antetul de forma (2) o vom numi *funcţie const*.

La apelul funcţiei care nu este *funcţie const*, pentru un obiect instanţiat cu ajutorul modificatorului *const*, compilatorul va emite un avertisment. Astfel, dacă vom utiliza funcţia *afsir* pentru clasa *sir* fără a fi definită ca *funcţie const*, atunci la apelul:

```
s1.afsir();
```

compilatorul afişează un avertisment.

Dacă modificăm antetul şi prototipul funcţiei *afsir* adăugînd modificatorul *const* după paranteza închisă:

```
void sir::afsir() const
```

atunci la apelul funcţiei *afsir* nu se va mai afişa avertismentul respectiv.

Se recomandă ca toate funcţiile membru care nu modifică nici o dată membru a obiectului curent, să fie realizate ca *funcţii const*.

26. OPERATORII DE DEREFERENŢIERE A POINTERILOR SPRE MEMBRII CLASELOR (.* ŞI ->*)

Aşa cum s-a văzut în capitolul 20, în limbajul C++, operatorul unar & (operatorul adresă) se utilizează nu numai pentru a defini adresa operanzilor, ci şi pentru a defini date de *tip referinţă*, date care permit realizarea *apelului prin referinţă* (call by reference). De aceea, în limbajul C++, operatorul unar & se mai numeşte şi *operator de referenţiere*.

Operatorul unar * (operatorul de indirectare) se aplică la un operand de tip pointer şi el asigură accesul la data aflată în zona de memorie a cărei adresă de început este valoarea operandului respectiv. În felul acesta operatorul unar * se poate considera că are un efect invers operatorului unar &.

Exemplu:

```
int x;
int *p;    //p este un pointer spre întregi de tip int
```

Fie:

```
p=&x;
```

Prin această expresie, lui *p* i se atribuie adresa zonei de memorie alocată lui *x*.

La valoarea lui *x* avem acces folosind fie numele *x*, fie expresia:

```
*p
```

formată din operandul *p* la care se aplică operatorul unar *.

Înlocuid pe *p* în expresia *p cu &x ajungem la concluzia că expresia:

```
*&x
```

are aceeaşi valoare ca şi operandul *x*.

Operatorul unar *, din expresia de mai sus, are un efect opus faţă de operatorul unar &. El se mai numeşte şi *operator de dereferenţiere*.

În limbajul C++ se pot defini pointeri spre elementele membru ale claselor, pointeri spre datele membru, precum şi pointeri spre funcţiile

membru ale claselor.

Pointerii spre elementele membru ale claselor se declară folosind numele clasei şi operatorul de rezoluţie.

Fie clasa cu numele *nume_clasa* care are o dată membru de tip *tip*. Atunci pointerul *p* spre o dată membru a clasei *nume_clasa* de tip *tip*, se declară astfel:

(1) *tip nume_clasa::*p;*

În mod analog, se definesc pointeri spre funcţiile membru ale clasei. Fie, de exemplu, o funcţie membru de prototip:

tip nume_clasa::nume_functie(*lista tipurilor parametrilor formali*);

Pointerul *pf* spre funcţia de acest prototip se declară astfel:

(2) *tip(nume_clasa::*pf)(lista tipurilor parametrilor formali)*;

Aceste declaraţii sînt analoge cu declaraţiile de pointeri obişnuite. Diferenţa dintre ele constă în aceea că se utilizează numele clasei urmat de operatorul de rezoluţie.

Pointerilor de acest tip li se atribuie valori ca mai jos.

Fie *d* o dată membru a clasei *nume_clasa* de tip *tip*.

Atunci o expresie de forma:

(3) *p = &nume_clasa::d*

atribuie lui *p* adresa datei membru *d*.

În mod analog, pointerului *pf* i se atribuie o valoare folosind o expresie de forma:

(4) *pf = nume_clasa::nume_functie*

unde:

nume_functie - Este o funcţie membru a clasei *nume_clasa*.
- Are prototipul indicat mai sus.

Expresiile (3) şi (4) sînt acceptate în corpul funcţiilor membru ale clasei *nume_clasa*, precum şi în orice parte a programului, dacă elementele membru nu sînt protejate (*d* şi *nume_functie*).

Pointerii *p* şi *pf* se utilizează aplicînd *operatorii de dereferenţiere* ".*" şi "->*".

Ei se utilizează asemănător cu operatorii "." şi "->".

Operatorul *punct* permite accesul la o componentă a unei date de un tip

definit de utilizator. Numele datei structurate se află în stînga punctului, iar numele componentei în dreapta lui.

În mod analog, în stînga operatorului ".*" se află instanţierea clasei (un obiect al clasei), iar în dreapta lui,în locul numelui componentei, se află pointerul spre componenta respectivă.

De exemplu, expresia:

*obiect.*p*

unde:

obiect - Este o instanţiere a clasei:

nume_clasa adică *nume_clasa obiect*; .

p - Este pointerul declarat în (1) şi la care i se atribuie o valoare prin expresia (3) permite acces la componenta:

obiect.d

unde:

d - Este o componentă a clasei *nume_clasa*.

Expresia (3) atribuie pointerului *p* deplasamentul componentei *d*, care stabileşte poziţia ei în instanţierile clasei *nume_clasa*.

Cu alte cuvinte, pointerul *p* declarat în (1) şi la care i se atribuie o valoare prin expresia (3), permite înlocuirea expresiei:

obiect.d

cu

*obiect.*p*

În felul acesta, *d* este schimbat cu **p* şi apare ca o dereferenţiere a pointerului *p*.

Operatorul "->*" se aplică atunci cînd în partea stîngă se utilizează nu o instanţiere a clasei (un obiect), ci un pointer spre tipul definit de clasa respectivă. Astfel, dacă *pobiect* se declară ca un pointer spre tipul *nume_clasa*:

*nume_clasa *pobiect*;

atunci pointerul *p* declarat în (1) şi la care i se atribuie valoare prin expresia (3), permite înlocuirea expresiei:

pobiect -> d

cu

*pobiect ->*p*

Şi în acest caz *d* apare ca o dereferenţiere a pointerului *p*.

Operatorii ".*" şi "->*" pot fi utilizaţi pentru a apela funcţii membru folosind declaraţii de forma (2) şi la care li se atribuie valori prin expresii de forma (4).

Apelul:

obiect.nume_functie(...)

unde:

obiect - Este o instanţiere a clasei *nume_clasa*.

nume_functie - Este o funcţie membru a clasei *nume_clasa* de prototipul indicat mai sus; se poate exprima folosind pointerul *pf* astfel:

(*obiect.*pf*)(...)

În mod analog, apelul:

pobiect -> nume_functie(...)

unde:

pobiect - Este un pointer spre *nume_clasa*, se poate exprima cu ajutorul pointerului *pf*, astfel:

(*pobiect ->*pf*)(...)

În ambele cazuri, *nume_functie* este schimbat cu **pf* şi apare ca o dereferenţiere a pointerului *pf*.

Operatorii ".*" şi "->*" au aceeaşi prioritate care este imediat mai mică decît a operatorilor unari şi imediat mai mare decît a operatorilor multiplicativi (/,*-binar şi %). Ei se asociază de la stînga la dreapta ca majoritatea operatorilor limbajului C++.

Operatorul ".*" nu poate fi supraîncărcat. Avînd în vedere acest fapt, lista completă a operatorilor care nu pot fi supraîncărcaţi este următoarea:

. .* :: ? :

Exerciţii:

26.1 Tipul *complex* se defineşte mai jos cu următoarele funcţii membru:

- constructorul cu parametri impliciţi;
- *sum*: însumează părţile reale sau imaginare a două obiecte;
- funcţii care supraîncarcă operatorii binari + şi - şi operatorul unar

minus;

- *afiscomplex*: afişează componentele unui obiect de tip complex;
- *citdouble*: citeşte un număr de la intrarea standard;
- *citcomplex*: citeşte componentele unui obiect de tip complex.

FIŞIERUL BXXVI1.H

```
class complex {
    double real;
    double imag;

    void sum(complex c1,complex c2,
             double complex::*p);
    /* - p pointeaza spre partea reala sau imaginara a obiectelor
         complexe;
       - partile spre care pointeaza p ale obiectelor c1 si c2 se insumea-
         za si se atribuie obiectului curent. */

    static int citdouble(char *text,double& d);
    /* - afiseaza text;
       - citeste un numar de la intrarea standard;
       - returneaza:
              0 - la EOF;
              1 - altfel.
*/

 public:
    complex(double x=0,double y=0);
    /* constructor utilizat la initializare si conversie dintr-un tip
       predefinit in tipul complex */

    friend complex operator +(complex c1,
                              complex c2);
    // returneaza suma c1+c2

    complex operator -();
    // returneaza negativul obiectului curent

    friend complex operator -(complex c1,
                              complex c2);
    // returneaza suma c1-c2
```

```
    void afiscomplex() const;
    // afiseaza componentele obiectului curent

    int citcomplex();
    /* - citeste componentele obiectului complex curent;
       - returneaza:
              0 - daca s-a intilnit sfirsitul de fisier;
              1 - altfel. */
};
```

Mai jos, se definesc funcţiile membru şi prieten ale clasei *complex* într-un fişier de extensie cpp.

FIŞIERUL BXXVI1

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#ifndef __COMPLEX_H
#include "BXXVI1.H"
#define __COMPLEX_H
#endif

int complex::citdouble(char *text,double& d)
/* - afiseaza text;
   - citeste un numar de la intrarea standard si-l pastreaza in zona refe-
     rita de parametrul d;
   - returneaza:
         0 - la intilnirea sfirsitului de fisier;
         1 - altfel.
*/
{
 char t[255];
 double x;

 for(;;){
    if(text && *text)
       printf(text);
    if(gets(t)==0)
       return 0;  // EOF
```

```
      if(sscanf(t,"%lf",&x) == 1)
          break;
      printf("nu s-a tastat un numar\n");
      printf("se reia citirea\n");
  }
  d=x;
  return 1;
 } //sfirsit citdouble

 int complex::citcomplex()
 /* - citeste componentele obiectului curent;
    - returneaza:
        0 - la intilnirea sfirsitului de fisier;
         1 - altfel.
 */

 {
  if(citdouble("Partea reala=",real)==0)
      return 0;  //EOF
  if(citdouble("Partea imaginara=",imag)==0)
      return 0;  //EOF
  return 1;
 }

 inline void complex::afiscomplex() const
 // afiseaza componentele obiectului curent
 {
  printf("%g+i*(%g)\n",real,imag);
 }

 inline void complex::sum(complex c1,complex c2,
                          double complex::*p)
 /* - p pointeaza spre partea reala sau imaginara a obiectelor
      de tip complex;
    - partile obiectelor c1 si c2 spre care pointeaza p se
      insumeaza si se atribuie obiectului curent */
 {
  this ->* p = c1.*p+c2.*p;
 }
```

```
inline complex::complex(double x,double y)
/* constructor utilizat la:
      - initializarea obiectelor;
      - conversia din tip predefinit in tip complex.
*/
{
 real = x;
 imag = y;
}

complex operator +(complex c1,complex c2)
// returneaza c1+c2
{
 complex r;

 double complex::*prelimag;
 // se insumeaza partile reale

 prelimag = &complex::real;
 r.sum(c1,c2,prelimag);
 // se insumeaza partile imaginare

 prelimag = &complex::imag;
 r.sum(c1,c2,prelimag);
 return r;
}

complex complex::operator -()
// returneaza negativul obiectului curent
{
 complex r = *this;
 r.real = -r.real;
 r.imag = -r.imag;
 return r;
}

inline complex operator -(complex c1,complex c2)
// returneaza c1-c2
{
 return c1 + -c2;
}
```

Observaţie:

La supraîncărcarea operatorului + se utilizează un pointer spre datele membru ale clasei complex. Acesta s-a declarat în corpul funcţiei respective astfel:

```
double complex::*prelimag;
```

Apoi, el este încărcat pentru a pointa spre partea reală:

```
prelimag = &complex::real;
```

În continuare se apelează funcţia *sum* care însumează părţile reale ale obiectelor care se adună.

Pentru a însuma părţile imaginare ale aceloraşi obiecte, nu avem decît să modificăm valoarea pointerului *prelimag* aşa ca el să pointeze spre partea imaginară:

```
prelimag = &complex::imag;
```

şi apoi să apelăm din nou funcţia *sum*.

Menţionăm că, metoda de mai sus pentru supraîncărcarea operatorului +, este mai complexă decît cea directă, utilizată în exerciţiile din capitolul 23. Ea a fost considerată pentru a exemplifica modul în care se utilizează pointerii spre date membru ale unei clase.

Metoda devine utilă cînd clasa are mai multe componente care se prelucrează la fel printr-o funcţie membru de o complexitate mai mare decît funcţia *sum* din acest exemplu.

26.2 Să se scrie un program care citeşte numere complexe de la intrarea standard şi afişează suma lor.

Programul de faţă utilizează tipul *complex* implementat în exerciţiul precedent.

PROGRAMUL BXXVI2

```
#include "BXXVI1.CPP"

main() /* citeste numere complexe si afiseaza suma lor */
{
 complex s;    // se instantiaza numarul complex nul
 complex z;    // se utilizeaza pentru a citi un numar complex
```

```
    for(;;){
        if(z.citcomplex()==0)
            break;  //EOF
        s = s+z;
    }
    s.afiscomplex();
  }
```

27. CONCLUZII ASUPRA PROGRAMĂRII PRIN ABSTRACTIZAREA DATELOR

Limbajele de programare oferă utilizatorului posibilitatea de a folosi în programare date de diferite tipuri, numite *tipuri predefinite*.

De obicei, toate limbajele de programare conţin tipuri predefinite pentru lucrul cu numere întregi de diferite dimensiuni.

Alte tipuri predefinite, utilizate frecvent, sînt cele pentru lucrul pe caractere şi cu numere neîntregi.

Datele de un anumit tip predefinit se disting prin modul de reprezentare şi prin operaţiile care se pot efectua asupra lor.

De exemplu, în limbajele C şi C++ se disting date de tip *întreg* care se reprezintă prin *complement faţă de 2*, precum şi date de *tip neîntreg* care au reprezentare diferită, numită *flotantă*. Cu ambele tipuri de date se pot face operaţii de calcul, cum ar fi cele 4 operaţii aritmetice, operaţii de operaţie etc. Există însă şi operaţii specifice, cum este de exemplu calculul restului împărţirii întregi (operatorul %) sau lucrul pe biţi, care sînt interzise pentru datele de alte tipuri decît cele întregi.

Pe scurt, tipurile predefinite au predefinite atît reprezentarea datelor cît şi operaţiile care se pot executa asupra lor.

Tipul predefinit poate fi considerat ca o mulţime de date care au o aceeaşi reprezentare şi asupra cărora este definit un set de operaţii. De aici decurge faptul că utilizarea datelor de un tip predefinit este controlată de compilator, în sensul că orice încercare de a utiliza operaţii neadmise asupra datelor de un tip predefinit va fi semnalată la compilare printr-un mesaj de eroare. În felul acesta se asigură o protecţie a datelor de tipuri predefinite faţă de utilizarea eronată a acestora.

În general, la rezolvarea unei probleme intervin şi alte tipuri de date decît cele predefinite.

În cazul limbajelor de programare apărute înaintea limbajului Pascal, conceptele care nu corespund tipurilor predefinite, se exprimă de către programator folosind tipurile predefinite din limbajele respective.

De exemplu, calculele cu numere fracţionare sau complexe se pot realiza în orice limbaj folosind tipurile predefinite existente în limbajul respectiv. Astfel, numerele fracţionare se definesc cu ajutorul a două date de tip

întreg. Operaţiile cu numere de acest fel se realizează apelînd proceduri (funcţii) corespunzătoare, ceea ce este posibil pentru orice limbaj care suportă programarea procedurală.

Pentru a simplifica folosirea în programare a conceptelor care nu corespund direct tipurilor predefinite în limbajul utilizat, s-a ajuns la ideea de a oferi utilizatorului posibilitatea reprezentării simple în limbajul de programare a conceptelor respective.

Primul limbaj care a fost proiectat în această idee a fost limbajul Pascal.

În acest limbaj se pot defini tipuri noi de date folosind construcţia RECORD (înregistrare).

Această idee a fost preluată şi de autorii limbajului C. În limbajul C se pot defini tipuri noi de date cu ajutorul construcţiei *struct*. Tipurile introduse în acest fel au fost numite *tipuri utilizator*. Ele deşi simplifică realizarea de concepte cu ajutorul tipurilor predefinite ale limbajului, între tipurile utilizator şi cele predefinite există diferenţe mari.

În primul rînd, construcţia *struct* defineşte numai reprezentarea datelor tipului utilizator, nu şi operaţiile care se pot executa cu datele respective.

De exemplu, tipul *complex*, în limbajul C, se defineşte ca mai jos:

```
struct complex {
    double real;
    double imag;
};
```

În continuare, se pot declara date de tip *complex* în mod obişnuit:

```
struct complex c1,c2,...;
```

Aceasta înseamnă că fiecare din datele *c1,c2,...*, se compune din două componente de tip *double*. La aceste componente avem acces folosind operatorul punct:

c1.real şi *c1.imag* - componentele lui *c1*;
c2.real şi *c2.imag* - componentele lui *c2*;

şi aşa mai departe.

În rest nu se mai cunoaşte nimic despre tipul *complex* definit ca mai sus.

Utilizatorul poate defini funcţii pentru a realiza diferite operaţii cu datele de tip *complex*.

Între reprezentarea datelor de tip *complex* (2 componente de tip *double*) şi funcţiile care definesc operaţii asupra datelor de acest tip nu este specificată nici o legătură. De aceea, compilatorul nu are nici un control

asupra operaţiilor care se execută asupra datelor de tip *complex*. De asemenea, operatorii obişnuiţi nu pot fi utilizaţi în cazul datelor de tip *complex*. O expresie de forma:

c1+c2

unde:

c1 şi c2 - Sînt declaraţi ca mai sus, este eronată.

Un număr mic de operatori sînt predefiniţi şi pentru date de tip utilizator. Aşa de exemplu, operatorul unar & (operatorul adresă) poate fi utilizat pentru operanzi care sînt de tip utilizator. Deci expresiile:

&c1,&c2,...

sînt admise de compilator şi au ca valoare adresa de început a zonei alocate operandului la care se aplică operatorul adresă.

Pasul următor în dezvoltarea limbajelor de programare a fost acela de a apropia, pe cît posibil, comportamentul datelor de tip utilizator de cel al datelor de tip predefinit.

În acest scop este necesar, în primul rînd, să se stabilească legătura dintre reprezentarea datelor şi operaţiile care se pot executa asupra datelor respective.

În al doilea rînd, este util ca operaţiile cu aceste date să se poată exprima cu ajutorul operatorilor, ca în cazul datelor de tipuri predefinite. De exemplu, este de dorit ca suma dintre datele de tip complex *c1* şi *c2* să se exprime cu ajutorul expresiei:

```
c1+c2
```

decît printr-un apel de funcţie de forma:

```
adcomplex(c1,c2)
```

În al treilea rînd, este necesar să se definească conversii din tipuri predefinite într-un astfel de tip şi invers, iar uneori chiar între două tipuri care nu sînt predefinite. De asemenea, este util ca astfel de conversii să se poată realiza implicit, ca în cazul tipurilor predefinite, iar alteori să poată fi explicitate folosind expresii *cast*.

În al patrulea rînd, este util ca datele de un astfel de tip să poată fi iniţializate la declararea lor.

Aceste facilităţi permit utilizatorului să extindă tipurile predefinite ale limbajului cu tipuri noi care să aibă aceeaşi comportare ca şi tipurile predefinite.

Ideea de a defini legătura dintre reprezentarea datelor unui tip şi operaţiile care se pot executa asupra datelor respective se realizează relativ simplu prin generalizarea construcţiei *struct*. Nu avem decît să înşirăm, în construcţia *struct*, prototipurile funcţiilor care definesc operaţii asupra datelor tipului respectiv. Totuşi, acest lucru nu este suficient, deoarece nu se asigură interzicerea accesului la datele tipului respectiv în afara funcţiilor a căror prototipuri se indică la definirea tipului. Astfel, o expresie de forma:

```
c1.real = 1
```

poate fi scrisă în orice punct al programului în care este definit *c1* prin declaraţia de mai sus.

Pentru a înlătura acest inconvenient este necesar să poată fi *protejat* accesul la date. Acest lucru se realizează utilizînd *clasele* în locul construcţiei *struct* extinsă ca mai sus.

Clasa are acelaşi format ca şi construcţia *struct* extinsă cu prototipuri de funcţii, dar în plus asigură protecţia la utilizarea componentelor sale.

Clasa are două feluri de componente:

- date

şi

- funcţii.

Datele componente ale unei clase se numesc *date membru* ale clasei respective, iar funcţiile componente, *funcţii membru*.

Clasa şi construcţia *struct* extinsă cu prototipuri de funcţii sînt facilităţi care sînt admise în limbajul C++, ele neputînd fi utilizate în limbajul C.

Menţionăm că şi construcţia *union*, din limbajul C, poate fi extinsă în C++ cu prototipurile funcţiilor, ca şi construcţia *struct*.

Construcţiile *struct* şi *union* extinse nu asigură însă protecţie la utilizarea componentelor sale spre deosebire de *clase*.

Prin utilizarea claselor se face un prim pas spre apropierea tipurilor care nu sînt predefinite de tipurile predefinite.

Un tip predefinit am văzut că are două aspecte:

- reprezentarea datelor

şi

- setul de operaţii admise cu datele respective.

Tot aşa, tipurile definite cu ajutorul claselor au aceleaşi proprietăţi,

definindu-se ambele aspecte.

În ambele cazuri, reprezentarea datelor constituie aşa numita *parte de implementare* a tipului.

Setul de operaţii admis asupra datelor unui tip constituie *interfaţa* programului cu tipul respectiv.

Interfaţa defineşte utilizarea în program a datelor tipului respectiv.

Această parte a unui tip, de obicei, *nu este protejată*.Se obişnuieşte să se spună că este *publică*.

În schimb, reprezentarea datelor trebuie să fie *protejată*, în sensul că, datele membru se pot accesa numai în corpul funcţiilor membru, nu şi în afara lor.

În general, partea de implementare a unei clase trebuie să fie protejată, iar partea de interfaţă să fie publică (neprotejată).

În mod implicit, în cazul claselor toate elementele sale sînt protejate. Evident, acest lucru nu este util şi de aceea, s-au introdus *modificatori de protecţie* care permit utilizatorului să stabilească părţile protejate şi publice ale unei clase.

Amintim că, o construcţie *struct* extinsă este ca o *clasă* care are toate elementele publice. De aceea, se obişnuieşte să se spună că o construcţie *struct* extinsă este o *clasă* cu elemente neprotejate.

Modificatorii de protecţie pot fi utilizaţi şi în cazul construcţiei *struct* extinse şi atunci ea se poate echivala cu orice *clasă*.

Construcţia *union* extinsă este şi ea ca o clasă care are toate elementele publice. În cazul ei nu se mai pot utiliza modificatorii de protecţie pentru a proteja anumite elemente.

Protecţia datelor membru ale unui tip definit printr-o clasă permite compilatorului să controleze operaţiile care se execută cu datele tipului respectiv. Într-adevăr, utilizatorul poate executa operaţii asupra datelor unui astfel de tip numai apelînd funcţii membru ale interfeţei publice. De exemplu, definind tipul *complex* printr-o clasă a cărei reprezentare este protejată:

```
class complex { //date membru care sint protejate in mod implicit
    double real;
    double imag;
 public:
// urmeaza partea neprotejata
};
```

şi dacă definim datele c1, c2 de tip complex:

```
complex c1,c2;
```

atunci o expresie de forma:

```
c1.real+c2.real
```

este eronată dacă se scrie în afara unei funcţii membru, deoarece componenta *real* este o dată membru protejată.

Expresiile:

```
&c1
c1=c2
```

sînt corecte deoarece operatorii adresă şi de atribuire sînt predefiniţi şi pentru date care sînt de un tip care nu este predefinit. Amintim că atribuirea care se realizează în acest caz este numită *atribuire implicită* şi aceasta este o copiere care se realizează bit cu bit pentru date de acelaşi tip.

Expresiile:

```
c1.real = c2.real;
c1.imag = c2.imag;
```

realizează acelaşi lucru ca şi atribuirea:

```
c1=c2;
```

dar ele pot fi scrise numai în corpul unei funcţii membru datorită protecţiei componentelor *real* şi *imag*. Instrucţiunea de atribuire care nu utilizează componentele protejate respective poate fi scrisă oriunde în program.

O expresie de forma:

```
c1+c2
```

este eronată deoarece operatorul + nu este predefinit şi pentru date care sînt de un tip, altul decît cele predefinite.

Pentru a realiza suma dintre datele *c1* şi *c2*, este necesară o funcţie care să fie apelată cu datele *c1* şi *c2* şi care să returneze suma lor. Această funcţie trebuie să fie *neprotejată* pentru a putea fi apelată în orice punct al programului în care este nevoie de suma a două numere complexe. Menţionăm că o funcţie membru se apelează totdeauna pentru a prelucra o dată de tipul pentru care ea este funcţie membru. De exemplu, dacă funcţia membru pentru adunarea datelor complexe c1+c2 o numim *adcomplex*, atunci ea se apelează pentru primul operand al sumei, iar cel de al doilea operand va fi parametru. De aceea, funcţia *adcomplex* are prototipul:

```
complex adcomplex(complex);
```

Apelul funcţiei *adcomplex* pentru a realiza suma c1+c2, are formatul:

```
c1.adcomplex(c2)
```

Utilizarea operatorului *punct* în apelul funcţiilor membru este în conformitate cu utilizarea lui în expresii de forma:

```
c1.real
```

şi

```
c1.imag
```

În ambele cazuri, în stînga punctului se află o dată de tip complex, iar în dreapta punctului un membru al tipului complex.

Să observăm că interfaţa unui tip definit printr-o clasă ne permite să utilizăm datele de tipul respectiv fără a ne interesa de implementarea lui. Astfel, apelul de mai sus al funcţiei *adcomplex* nu conţine componentele dată (real şi imag) ale lui *c1* şi *c2*. Dimpotrivă, utilizatorului îi este interzis accesul direct la componentele respective.

În concluzie, datele unui tip definit printr-o clasă se prelucrează folosind numai interfaţa tipului respectiv, făcînd *abstracţie* de reprezentarea datelor tipului respectiv. În felul acesta, tipul definit printr-o clasă diferă substanţial faţă de tipurile utilizator care se definesc cu ajutorul construcţiei *struct* în limbajul C.

Tipurile definite cu ajutorul claselor se numesc *tipuri abstracte de date*.

Din cele de mai sus rezultă că, tipurile abstracte de date răspund primului deziderat al tipurilor de date care este şi cel mai important şi anume acela de a stabili legătura dintre reprezentarea datelor şi operaţiile care se pot executa asupra datelor respective.

Datele de tip abstract se mai numesc şi *obiecte*. Despre obiectele care sînt de un acelaşi tip abstract se spune că sînt *instanţieri* ale clasei care defineşte tipul respectiv.

Datorită protecţiei pe care o asigură clasele datelor membru, se obişnuieşte să se spună că datele respective sînt *încapsulate* în clase.

Menţionăm că se pot proteja şi funcţii membru. În acest caz, funcţiile respective nu pot fi apelate direct din program, ci numai din corpul altor funcţii membru.

De asemenea, se pot defini date membru publice (neprotejate), dar se recomandă evitarea situaţiilor de acest fel.

Obiectul pentru care se apelează o funcţie membru se numeşte *obiectul curent*. În corpul funcţiei membru se poate face acces direct, fără nici o

restricţie, la componentele obiectului curent.

Uneori este nevoie să ne referim la obiectul curent (nu la componente ale acestuia). Acest lucru este posibil utilizînd pointerul implicit *this*. Acesta este definit în corpul oricărei funcţii membru (nestatică) şi are ca valoare adresa obiectului curent.

De exemplu, în momentul apelului:

```
c1.adcomplex(c2)
```

în corpul funcţiei *adcomplex*, pointerul *this* are ca valoare adresa lui *c1* (&c1). De fapt, acest pointer se aplică în mod implicit în corpul funcţiilor membru nestatice la componentele obiectului curent.

De exemplu, funcţia *adcomplex*, de mai sus,se defineşte astfel:

```
complex complex::adcomplex(complex c)
{
 complex r;   //obiect temporar
 r.real = real + c.real;
 r.imag = imag + c.imag;
 return r;
}
```

Elementele *real* şi *imag* utilizate fără operatorul *punct* sînt componentele obiectului curent. Lor li se aplică, în mod implicit de către compilatorul C++, operatorul -> folosind pointerul *this*. Deci,instrucţiunile respective sînt traduse de compilator ca şi cînd ar fi scrise sub forma:

```
r.real = this -> real + c.real;
r.imag = this -> imag + c.imag;
```

Funcţiile membru *statice* (declarate cu ajutorul cuvîntului *static* în definiţia clasei) nu se apelează pentru un obiect al clasei. De aceea, în acest caz nu există noţiunea de obiect curent şi nici pointerul *this* nu este definit în corpul lor.

Cel de-al doilea deziderat, ca să se poată utiliza operatorii obişnuiţi ai limbajului cu operanzi obiecte, este şi el realizat în limbajul C++. În acest caz se porneşte de la noţiunea de *supraîncărcare a funcţiilor* (supradefinire, redefinire etc.).

În limbajul C++ se admite ca mai multe funcţii să aibă acelaşi nume, funcţii care să difere între ele prin numărul şi tipurile parametrilor formali.

Operatorii din limbajele C şi C++ se pot aplica la date de diferite tipuri predefinite. De exemplu, expresia:

```
d1+d2
```

este corectă atît pentru date de tip întreg (*int, long,unsigned*), cît şi pentru date neîntregi (de tip *float, double* şi *long double*). Se spune că operatorul + este supraîncărcat pentru tipurile *int, long, unsigned, float, double* şi *long double*. Această supraîncărcare este predefinită. Pentru a putea utiliza expresia de mai sus şi în cazul cînd operanzii *d1* şi *d2* sînt obiecte (date de tipuri abstracte) este necesar a supraîncărca operatorul + pentru tipurile obiectelor respective. Aceasta se realizează simplu folosind funcţii membru, de nume special, pentru ca compilatorul să recunoască sensul funcţiilor respective.

O astfel de funcţie are numele format din cuvîntul cheie *operator* urmat de operatorul pe care-l supraîncarcă. În rest, corpul funcţiei este obişnuit. De exemplu, pentru a supraîncărca operatorul + pentru obiecte de tip *complex*, în locul funcţiei *adcomplex* de mai sus, vom folosi funcţia definită astfel:

```
complex complex::operator +(complex c)
{
 complex r;

 r.real = real + c.real;
 r.imag = imag + c.imag;
 return r;
}
```

La întîlnirea unei expresii de forma:

```
c1+c2
```

unde *c1* şi *c2* sînt obiecte de tip *complex*, compilatorul apelează, în mod automat, funcţia membru *operator* +, care are acelaşi efect cu apelul funcţiei *adcomplex* pentru obiectul *c1* şi cu parametrul *c2*:

```
c1.adcomplex(c2)
```

De altfel, compilatorul, la întîlnirea expresiei de mai sus, realizează în mod automat apelul:

```
c1.operator +(c2)
```

Se pot supraîncărca numai operatorii existenţi din limbajul C++, exceptînd operatorii:

. .* :: ? :

La supraîncărcarea operatorilor se poate proceda ca mai sus, folosind funcţii membru nestatice. Totuşi, există cîţiva operatori care prezintă reguli specifice la supraîncărcarea lor. Aceştia sînt:

->, [], (), new,delete, operatorul *cast*, =, +=, -=, *=,
/=, &=, ^=, |=, <<= şi >>=
(aici virgula a fost folosită ca separator).

De exemplu, operatorii *new* şi *delete* se supraîncarcă prin funcţii *membru statice*, iar ceilalţi prin funcţii membru nestatice. Amănunte în legătură cu supraîncărcarea operatorilor de mai sus se dau în capitolul 23, exceptînd operatorul *cast* care este tratat în capitolul 24.

Funcţiile membru nestatice prezintă unele inconveniente la supraîncărcarea operatorilor. Într-adevăr, dacă expresia:

```
c1+c2
```

poate fi tratată supraîncărcînd operatorul + ca mai sus, expresiile:

```
c1+d
```

şi

```
d+c2
```

unde *d* este o dată de tip predefinit (*int*, *long*, *double* etc.) nu sînt recunoscute pe baza supraîncărcării operatorului + cu ajutorul funcţiei *operator* + de mai sus.

Expresia:

```
c1+d
```

poate fi tratată definind, ca mai jos, încă o funcţie membru nestatică pentru supraîncărcarea operatorului +:

```
complex complex::operator+(double d)
{
 complex r;

 r.real = real+d;
 r.imag = imag;
 return r;
}
```

În schimb, expresia:

```
d+c2
```

nu se poate trata supraîncărcînd operatorul + cu funcţii membru nestatice. Aceasta deoarece o funcţie membru nestatică se apelează totdeauna pentru un obiect care este o instanţiere a clasei pentru care funcţia respectivă este funcţie membru, iar pe de altă parte, funcţiile *operator* nestatice se apelează totdeauna pentru obiectul din stînga operatorului pe care-l supraîncarcă.

În cazul expresiei de mai sus, nu se poate defini o funcţie operator + care să fie funcţie membru nestatică pentru clasa *complex*, deoarece în stînga operatorului + se află o dată care nu este de tip *complex*.

Pentru a putea utiliza expresii de forma celei de mai sus (primul operand este de un tip predefinit), operatorul + se supraîncarcă folosind o *funcţie prieten* a clasei *complex*.

Funcţiile *prieten* pentru o clasă sînt funcţii obişnuite sau funcţii membru ale unei alte clase, care au acces la elementele protejate ale clasei respective, la fel ca şi funcţiile membru ale acelei clase.

Pentru a declara că o funcţie este o funcţie prieten a unei clase, prototipul funcţiei respective se scrie precedat de cuvîntul *friend* în definiţia acelei clase.

În cazul clasei *complex* se procedează astfel:

```
class complex {
     double real;
     double imag;
 public:
     ...
     complex operator +(complex c);
     // ambii operanzi sint de tip complex; functie membru nestatica

     complex operator +(double d);
     // primul operand este complex, iar al doilea de un tip predefinit
     // functie membru nestatica

     friend complex operator +(double d,complex c);
     // returneaza suma d+c
     // functie prieten a clasei complex; are acces la elementele
     // protejate: real si imag
     ...
};
```

Funcţia prieten pentru supraîncărcarea operatorului + se poate defini astfel:

```
complex operator +(double d,complex c)
// returneaza suma d+c
{
 return c+d;
}
```

Funcţiile prieten au şi alte utilizări decît cele legate de supraîncărcarea

operatorilor.

O utilizare importantă a funcţiilor prieten este cea cu privire la construirea *iteratorilor*. În acest caz, iteratorul se realizează printr-o clasă specială care este o *clasă prieten* a clasei pentru care se implementează iteratorul respectiv (vezi supraîncărcarea operatorului ()).

O clasă *cl1* este o clasă prieten pentru clasa *cl2*, dacă se declară ca atare (`friend class cl1;`) în definiţia clasei *cl2*.

Dacă o clasă *cl1* este o clasă prieten a clasei *cl2*, atunci toate funcţiile membru ale clasei *cl1* sînt funcţii prieten pentru clasa *cl2*.

Cel de-al treilea deziderat pentru tipurile abstracte şi anume acela ca să se poată defini conversii pentru obiecte care să se aplice implicit sau explicit, este şi el implementat în limbajul C++ şi se poate realiza prin intermediul supraîncărcării operatorului *cast* (vezi capitolul 24), precum şi cu ajutorul *constructorilor*.

Constructorii sînt nişte funcţii membru speciale care au importanţă mai ales în legătură cu iniţializarea obiectelor. Ei au acelaşi nume cu numele clasei pentru care se definesc.

Pentru detalii în legătură cu constructorii se va vedea capitolul 22.

Constructorii se pot utiliza pentru a realiza conversii dintr-un tip predefinit în tipul abstract definit de clasa pentru care sînt funcţii membru (vezi capitolul 24).

Un astfel de constructor are un singur parametru de tip predefinit. De exemplu, pentru a realiza conversii dintr-un tip predefinit în tipul *complex* se poate folosi următorul constructor:

```
complex::complex(double d)
{
 real = d;
 imag = 0;
}
```

Conversiile inverse, dintr-un tip abstract într-un tip predefinit, se realizează pe baza supraîncărcării operatorului *cast*.

De exemplu, conversia obiectelor de tip *complex* în tipul *double* se poate realiza folosind funcţia de mai jos pentru supraîncărcarea operatorului *cast*:

```
complex::operator double()
{
 return real;
}
```

Aceste conversii uneori se aplică implicit, iar în alte cazuri este necesar să se aplice explicit operatorul *cast* supraîncărcat.

Conversiile realizate cu ajutorul constructorilor sau a operatorului *cast* supraîncărcat se numesc *conversii definite de utilizator* sau mai scurt *conversii utilizator*.

Ca regulă generală, conversiile utilizator implicite se aplică în ultimul rînd în evaluarea expresiilor, adică atunci cînd expresia nu poate fi evaluată pe baza aplicării de conversii implicite pentru date predefinite şi pe baza apelării funcţiilor care supraîncarcă operatorii pentru operanzii la care se aplică. Aceasta înseamnă că, conversiile utilizator sînt cele mai puţin prioritare.

Exemple:

Fie clasa *complex*, definită ca mai jos:

```
class complex {
    double real;
    double imag;
  public:
    complex(double x,double y=0)  // constructor
    {
        real=x;
        imag=y;
    }
    complex operator +(complex c)
    // supraincarcarea operatorului +
    {
        complex r;
        r.real = real+c.real;
        r.imag = imag+c.imag;
        return r;
    }
    operator double()
    // supraincarca operatorul cast
    {
        return real;
    }
    ...
};
```

Constructorul clasei *complex* realizează conversii dintr-un tip predefinit, în obiecte de tip *complex*. El se apelează în mod automat la instanţierea obiectelor:

```
complex z1(1,2);    //se instantiaza z1=1+2i
complex z2(3,4);    //se instantiaza z2=3+4i
complex z3(0);      //se instantiaza z3=0+0i
```

Expresia de atribuire:

```
z3=z1+z2
```

este corectă şi se evaluează apelînd funcţia care supraîncarcă operatorul "+":

```
z3 = z1.operator +(z2)
```

La revenirea din funcţie se returnează un obiect care corespunde sumei dintre obiectele complexe *z1* şi *z2* şi care, în lipsa supraîncărcării operatorului "=", se atribuie bit cu bit lui *z3*.

O expresie de forma:

```
z3 = z1+123
```

este şi ea corectă. Într-adevăr, în acest caz se aplică automat secvenţa de conversii:

123
- Se converteşte din *int* în *double* (conversie implicită dintr-un tip predefinit într-un alt tip predefinit).
- Rezultatul conversiei precedente se converteşte din *double* în *complex* (conversie implicită din *double* în tipul abstract *complex* realizată pe baza apelului constructorului).

Acest şir de conversii implicite rezultă din faptul că operatorul + este supraîncărcat cu o funcţie membru nestatică a clasei *complex* precum şi din modul de interpretare a expresiei de mai sus. Astfel, expresia de mai sus conduce la următorul apel al funcţiei *operator* +:

```
z3 = z1.operator +(123)
```

Deoarece funcţia *operator* + are un parametru de tip *complex*, parametrul efectiv 123 este supus conversiilor implicite amintite mai sus.

Să presupunem că, clasa *complex* definită mai sus,mai are încă o funcţie membru nestatică pentru supraîncărcarea operatorului "+" definită ca mai jos:

```
complex complex::operator +(double d)
// returneaza obiectul curent adunat cu d
{
 complex r;
 r.real = real+d;
 r.imag = imag;
 return r;
}
```

În acest caz, expresia:

```
z3 = z1+123
```

se realizează fără a mai aplica conversia implicită realizată prin constructor.

Expresia se evaluează apelînd funcţia de mai sus care are un parametru formal de tip *double*:

```
z3 = z1.operator(123)
```

Parametrul efectiv 123 se converteşte spre *double* şi apoi se atribuie lui *d*. Acest mod de interpretare decurge din faptul că, conversia realizată prin constructor este o conversie utilizator şi ea este mai puţin prioritară faţă de apelul funcţiilor care supraîncarcă operatorii.

În condiţiile de mai sus, expresia:

```
z3 = 123+z1
```

este eronată. Aceasta deoarece, în acest caz, expresia se interpretează astfel:

```
z3 = 123.operator(z1)
```

Cu alte cuvinte, într-o astfel de expresie nu se mai aplică în mod implicit constructorul pentru a realiza conversia lui 123 într-un obiect de tip complex.

O soluţie posibilă este aplicarea explicită a constructorului:

```
z3 = complex(123)+z1
```

O altă soluţie posibilă este secvenţa:

```
z3 = 123;
z3 = z3+z1
```

Prima instrucţiune conduce la o conversie a numărului 123 în *double* (conversie dintr-un tip predefinit în altul), iar din *double* în tipul *complex* prin apelul implicit al constructorului.

O altă soluţie posibilă este de a defini o funcţie prieten pentru supraîncărcarea operatorului "+", aşa cum s-a indicat mai sus.

Prezenţa constructorului care să realizeze conversia dintr- un tip predefinit în tipul *complex* permite însă o soluţie mult mai simplă, cu o singură funcţie prieten pentru supraîncărcarea operatorului "+". Într-adevăr, fie definiţia clasei *complex* de mai jos:

```
class complex {
     double real;
     double imag;
 public:
     complex(double x,double y=0)
     {
         real = x;
         imag = y;
     }

     friend complex operator +(complex z1,
                               complex z2)
     // returneaza z1+z2
     {
         complex r;
         r.real = z1.real + z2.real;
         r.imag = z1.imag + z2.imag;
         return r;
     }

     operator double()
     // conversie din complex in double
     {
         return real;
     }
     ...
};
```

Considerăm instanţierile *z1, z2* şi *z3* ca mai sus, adică:

```
complex z1(1,2);
complex z2(3,4);
complex z3(0);
```

Supraîncărcarea operatorului "+", împreună cu constructorul clasei *complex* asigură recunoaşterea şi tratarea corectă a expresiilor de mai jos:

```
z3 = z1+z2;     // nu se face conversie
z3 = z1+123;    // 123 se converteste spre tipul complex
z3 = 123+z2;    // 123 se converteste spre tipul complex
```

Expresiile de mai sus sînt interpretate astfel:

```
z3 = operator +(z1,z2);  /* apel in care parametrii efectivi
                            au acelasi tip cu cei formali */
z3 = operator +(z1,123); /* cel de al doilea parametru
                            efectiv se converteste spre
                            tipul complex */
z3 = operator +(123,z2); /* primul parametru efectiv se
                            converteste spre tipul complex */
```

Această soluţie pare a fi cea mai simplă şi se poate utiliza la supraîncărcarea operatorilor binari obişnuiţi. Ea presupune două lucruri:

- existenţa constructorului care să permită conversia dintr-un tip predefinit în tipul abstract pentru care se defineşte supraîncărcarea operatorilor binari obişnuiţi;
- supraîncărcarea fiecărui operator binar obişnuit printr-o funcţie prieten ai cărei parametri formali sînt ambii de tipul abstract pentru care se face supraîncărcarea.

Fie declaraţia:

```
double x;
```

O expresie de atribuire de forma:

```
x=z1
```

unde *z1* este un obiect *complex* declarat ca mai sus, este corectă şi atribuie lui *x* partea reală a lui *z1*.

Această atribuire se realizează pe baza aplicării implicite a funcţiei membru care supraîncarcă operatorul *cast* pentru conversia obiectelor complexe într-o dată de un tip predefinit.

Variabila *x* poate avea orice tip predefinit. De exemplu, dacă *y* este de tip *long*:

```
long y;
```

atunci expresia:

```
y = z1
```

implică 2 conversii:

- una din *complex* în *double*

şi apoi

- din *double* în *long*.

Să observăm că o expresie de forma:

```
x = z1+123
```

unde *x* şi *z1* sînt declarate ca mai sus, se evaluează astfel:

1. Se apelează funcţia care supraîncarcă operatorul "+" pentru a realiza suma. Acest apel implică conversia 123 în *double* şi apoi în *complex* prin apelul implicit al constructorului.
2. Se apelează operatorul *cast* supraîncărcat pentru a converti suma complexă într-o dată de tip *double*.
3. Se atribuie valoarea rezultată la punctul 2, variabilei *x*.

O evaluare mai economică se realizează dacă obiectul *z1* se converteşte în *double* şi abea apoi se realizează adunarea, după regulile obişnuite.

În acest scop, se poate folosi secvenţa:

```
x = z1;
x += 123;
```

O altă posibilitate este aceea de a explicita conversia lui *z1* spre *double* prin utilizarea explicită a operatorului *cast*:

```
x = (double)z1+123
```

sau

```
x = double(z1)+123
```

Conversia dintr-un tip abstract într-un alt tip abstract se poate realiza fie cu ajutorul unui constructor adecvat, fie supraîncărcînd operatorul *cast* în mod corespunzător.

Fie *t1* şi *t2* două tipuri abstracte. Pentru a face conversia obiectelor din tipul *t1* în tipul *t2*, se poate defini pentru clasa *t2* un constructor de prototip:

```
t2(t1);
```

Pentru ca constructorul de acest prototip să poată realiza conversia obiectelor din tipul *t1* în tipul *t2*, de obicei, trebuie să aibă acces la datele membru protejate ale clasei *t1*. De aceea, constructorul de mai sus se va declara funcţie prieten a clasei *t1*:

```
friend t2::t2(t1);
```

Un exemplu de astfel de constructor s-a definit în exerciţiul 24.15 pentru conversia obiectelor din tipul *dc* în tipul *dczi*. De asemenea, clasa *dc* dispune de un constructor care realizează conversia inversă.

Aşa cum s-a amintit mai sus, conversia din tipul *t1* în tipul *t2* se poate

realiza şi prin supraîncărcarea operatorului *cast*.

În acest caz, clasa *t1* va avea o funcţie membru nestatică de antet:

```
operator t2();
```

Funcţia respectivă, de obicei, trebuie să aibă acces la datele membru protejate ale clasei *t2* şi de aceea, ea se va declara ca funcţie prieten a clasei *t2*:

```
friend t1::operator t2();
```

Un exemplu de supraîncărcare a operatorului *cast* se dă în exerciţiul 24.12. În acest exerciţiu se supraîncarcă operatorul *cast* pentru a face conversii din grade *fahrenheit* în grade *celsius*.

Menţionăm că, prezenţa mai multor posibilităţi de conversii dintr-un tip abstract într-un alt tip abstract (conversie atît prin constructor, cît şi prin supraîncărcarea operatorului *cast*) poate conduce la ambiguităţi.

Utilizarea constructorilor la conversii este o metodă folosită frecvent şi de obicei ea se aplică în mod implicit.

Uneori aplicarea implicită a conversiilor pe baza apelului implicit al constructorilor poate să conducă la situaţii nedorite sau mai puţin eficiente. Un astfel de exemplu este evaluarea expresiei de mai jos:

```
x = z1+123
```

unde operandul 123 se converteşte spre tipul *complex*, în loc să se procedeze la conversia lui *z1* în *double*.

În legătură cu aplicarea conversiilor definite de utilizator, amintim că dacă la evaluarea unui operand este nevoie să se realizeze mai multe conversii implicite, atunci dintre acestea numai una poate implica tipuri abstracte, adică se admite o singură conversie utilizator, celelalte putînd fi numai predefinite.

Cel de-al patrulea deziderat se referă la iniţializarea obiectelor. Aceasta se realizează cu ajutorul constructorilor. Aceştia sînt funcţii membru care se apelează automat la declararea obiectelor.

Datele de iniţializare sînt valori pentru parametrii formali ai constructorilor.

Se pot defini mai mulţi constructori pentru aceeaşi clasă, care să difere prin numărul şi/sau tipul parametrilor.

În cazul în care nu există nici un constructor definit de utilizator pentru o clasă, compilatorul generează în mod automat un constructor pentru clasa respectivă. Acest constructor nu are parametri. El nu serveşte la

iniţializarea obiectelor clasei respective, ci are rol numai la alocarea obiectelor.

Un constructor fără parametri se numeşte constructor *implicit*.

În cazul în care clasa are cel puţin un constructor definit de programator, compilatorul nu mai generează un constructor pentru clasa respectivă.

În concluzie, clasa poate să nu aibă constructori dacă nu se doreşte iniţializarea obiectelor la instanţierea lor şi în acest caz compilatorul generează un constructor implicit.

Constructorul implicit se defineşte de programator în cazul în care sînt definiţi şi alţi constructori pentru clasa respectivă care au fiecare cel puţin un parametru neimplicit (fără valoare implicită) şi se doreşte instanţierea de obiecte neiniţializate.

Dacă o clasă are definit un constructor care are toţi parametri impliciţi, atunci pentru el nu se mai poate defini şi un constructor implicit, deoarece apelul acestora devine ambigu.

Exemplu:

```
class complex {
     double real;
     double imag;
 public:
     complex()  //constructor implicit
     {
     }

     complex(double x=0,double y=0)
    // constructor cu toti parametri impliciti
     {
         real = x;
         imag = y;
     }
     ...
};
```

Declaraţia:

```
complex z;
```

devine ambiguă, deoarece la instanţierea lui *z* se poate aplica oricare dintre cei doi constructori ai clasei *complex*. În astfel de situaţii se renunţă la constructorul implicit.

Clasa *complex* definită ca mai jos:

```
class complex {
    double real;
    double imag;
  public:
    complex(double x,double y=0)
    {
        real=x;
        imag=y;
    }
    ...
};
```

permite instanţierea numai de obiecte iniţializate.

Pentru a putea declara obiecte neiniţializate, va fi nevoie să se definească constructorul implicit.

Aşa cum s-a arătat mai sus, constructorii unei clase se pot utiliza la conversiile datelor de tip predefinit în obiecte ale clasei respective. În acest caz, constructorul trebuie să aibă cel puţin un parametru. Dacă sînt mai mulţi, cel mult unul poate să nu fie implicit.

Parametrii constructorului pot fi de orice tip, dar care este diferit de tipul abstract implementat de clasa pentru care este constructor.

Totuşi, parametrii unui constructor pot fi pointeri sau referinţe la tipul abstract implementat de clasa la care aparţine constructorul.

Exemplu:

Un constructor de prototip:

complex::*complex*(*complex z*);

este eronat. În schimb, se poate defini un constructor de următorul prototip:

complex::*complex*(*complex*& *z*);

Un constructor de prototip:

nume_clasa::*nume_clasa*(**const** *nume_clasa*&);

se numeşte *constructor de copiere*.

Acesta se apelează automat în următoarele situaţii:

- la instanţierea unui obiect al clasei *nume_clasa* se utilizează, pentru iniţializare, un alt obiect al aceleeaşi clase şi care deja este instanţiat;
- la transferul parametrilor efectivi care sînt obiecte ale clasei

nume_clasa, transfer care nu implică conversii utilizator;

- la revenirea dintr-o funcţie care returnează un obiect al clasei *nume_clasa*.

Să observăm că, constructorul de copiere nu se apelează la evaluarea atribuirilor.

Dacă pentru o clasă nu există definit un constructor de copiere, atunci compilatorul generează în mod automat un astfel de constructor pe care îl numim *constructor de copiere implicit*. Acesta realizează o copiere bit cu bit, ceea ce adesea este suficientă. De exemplu, pentru tipul *complex*, implementat ca în exemplele precedente este suficientă copierea bit cu bit.

În general, constructorul de copiere este necesar pentru clase care au componente date de tip pointer sau referinţă.

Un exemplu de astfel de clasă este clasa *sir*, implementată în exerciţiul 23.14.

Dacă constructorul se apelează la crearea obiectelor, la dezalocarea (distrugerea) lor se apelează o altă funcţie membru numită *destructor*.

În general nu este obligatoriu să definim un destructor pentru o clasă. Dacă o clasă nu are definit un destructor de către programator, atunci compilatorul generează în mod automat un destructor pentru clasa respectivă.

Destructorul generat de compilator se numeşte *destructor implicit*.

Destructorii sînt funcţii membru speciale care nu au parametri (vezi capitolul 22).

Pentru o clasă se poate defini cel mult un destructor.

Destructorii se definesc, de obicei, pentru clase care au date membru pointeri sau referinţe. Un exemplu de astfel de clasă este clasa *sir* din exerciţiul 23.14.

Programarea care se bazează pe utilizarea tipurilor abstracte de date o vom numi *programare prin abstractizarea datelor*.

În principiu, rezolvarea unei probleme folosind programarea prin abstractizarea datelor presupune ca prim pas, stabilirea conceptelor care intervin în descrierea metodei de rezolvare a problemei respective. La pasul următor se implementează conceptele care nu corespund tipurilor predefinite ale limbajului ales pentru rezolvarea problemei. Dacă limbajul utilizat este limbajul C++, atunci conceptele care nu corespund tipurilor predefinite se implementează sub formă de tipuri abstracte folosind clase. Apoi, se descrie metoda de rezolvare a problemei respective folosind

tipurile abstracte implementate prin clase ca şi cum tipurile respective ar exista implementate în limbajul C++ alături de cele predefinite.

Dintre conceptele utilizate mai frecvent în programare şi care nu sînt implementate ca tipuri predefinite în limbajul C++ amintim:

- şiruri de caractere;
- numere raţionale;
- numere complexe;
- tablouri dinamice;
- liste simplu înlănţuite;
- liste dublu înlănţuite;
- liste circulare;
- arbori;
- tabele de dispersie.

Un neajuns al programării prin abstractizarea datelor este faptul că nu permite exprimarea legăturilor dintre diferite concepte. Singura legătură dintre concepte care se poate exprima, este aceea că datele membru ale unei clase pot fi obiecte ale unei alte clase. Acest lucru nu este suficient în cazul în care conceptele sînt strîns dependente între ele formînd structuri "ierarhice".

Exprimarea ierarhiilor conduce la atribute suplimentare cum sînt cele de *moştenire*. Aceasta conduce la un model nou de programare pe care îl numim *programare orientată spre obiecte*.

28. PROGRAMAREA ORIENTATĂ SPRE OBIECTE

Istoria dezvoltării ştiinţelor marchează o perioadă lungă în care s-au acumulat date despre formele de existenţă ale materiei şi fenomenele care au loc în natură şi în viaţa noastră de zi cu zi.

Perioada următoare se caracterizează prin aceea că, datele acumulate au fost supuse unui proces de clasificare potrivit unor trăsături care se constată că sînt comune pentru mai multe fenomene sau forme ale materiei moarte sau vii. Acest proces conduce în mod natural la stabilirea de *ierarhii* între diferitele fenomene şi forme de existenţă ale materiei.

În vîrful unei ierarhii se află fenomenul sau forma de existenţă care are trăsături *comune* pentru toate celelalte componente ale ierarhiei respective. Pe nivelul următor al ierarhiei se află componentele care pe lîngă trăsăturile comune de pe nivelul superior, mai au şi trăsături suplimentare, *specifice*. O ierarhie, de obicei, are mai multe nivele, iar situarea unui element pe un nivel sau altul al ierarhiei este uneori o problemă deosebit de complexă.

Elementul din vîrful ierarhiei este considerat a fi cel mai *general* dintre elementele ierarhiei. Toate celelalte elemente ale ierahiei sînt *mai particulare, mai specializate*. Acest lucru este valabil în orice punct al ierahiei. Astfel, un element aflat la un nivel al ierarhiei este mai particular (mai specializat) decît cel aflat pe nivelul imediat anterior lui şi este mai general decît elementul aflat pe un nivel imediat următor lui. Se obişnuieşte să se spună că un element al ierarhiei *moşteneşte* trăsăturile elementelor de pe nivelele anterioare.

În general, un element al ierarhiei poate moşteni trăsături de la mai multe elemente ale ierarhiei. Ierarhia este *arborescentă*, dacă orice element al ei, diferit de elementul din vîrful ierarhiei, moşteneşte trăsăturile de la un singur element.

La ora actuală, toate ramurile cunoaşterii ştiinţifice sînt pline de ierarhii rezultate în urma clasificării cunoştinţelor acumulate în perioada lungă de observare a fenomenelor şi formelor de existenţă a lumii materiale. Astfel, clasificările ierarhice ale cunoştinţelor pot fi întîlnite atît în domeniile care pleacă de la cele mai concrete forme ale lumii materiale, cum sînt botanica, zoologia, biologia etc., cît şi în domeniul care se consideră că studiază conceptele cele mai abstracte, cum sînt ramurile matematicii: geometria, algebra abstractă etc.

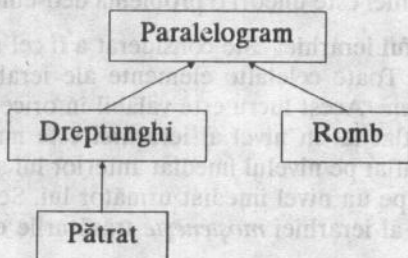
Un exemplu simplu întîlnit în geometrie a fost amintit chiar în

introducerea acestei cărţi. El se referă la clasificarea patrulaterelor. Astfel, dacă se consideră că *paralelogramul* este în vîrful ierarhiei, atunci pe nivelul următor se poate aşeza *dreptunghiul*. Într-adevăr, dreptunghiul este un paralelogram la care se adaugă condiţia, să aibă un unghi drept. În felul acesta, dreptunghiul are (moşteneşte) toate proprietăţile paralelogramului, dar mai are şi proprietăţi în plus (de exemplu, diagonalele dreptunghiului sînt egale). Paralelogramul este un concept mai general decît dreptunghiul, iar acesta din urmă este un paralelogram particular.

În mod analog, *pătratul* se poate defini ca fiind un dreptunghi particular şi anume un dreptunghi care are toate laturile egale. Aceasta înseamnă că pătratul moşteneşte toate proprietăţile dreptunghiului, dar faţă de acesta, pătratul mai are şi alte proprietăţi (de exemplu, diagonalele lui sînt axe de simetrie).

Rombul este şi el un paralelogram particular şi anume, el este un paralelogram care are toate laturile egale. El moşteneşte toate proprietăţile paralelogramului dar mai are şi proprietăţi în plus (de exemplu, are diagonalele perpendiculare).

Se obţine în felul acesta o ierarhie arborescentă ca în figura de mai jos.



Un alt exemplu simplu de ierarhie este ierarhia stabilită între conceptele *grupoid, semigrup, monoid, grup, inel* şi *corp* din algebra abstractă. Această ierarhie este schematizată ca mai jos.

Să amintim pe scurt definiţiile acestor concepte din algebra abstractă.

Ca noţiune primară vom considera noţiunea de *mulţime*.

Fie A şi B două mulţimi. Cu AxB se notează mulţimea perechilor ordonate de elemente:

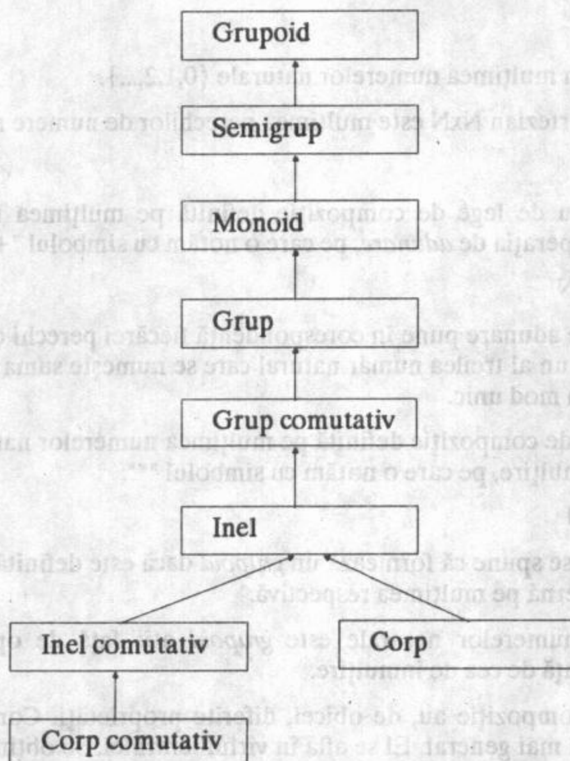
(x,y)

unde:

x - Este element al mulţimii A.

y - Este element al mulţimii B.

Mulţimea AxB se numeşte *produsul cartezian* al mulţimilor A şi B.



Fie A şi B două mulţimi. Dacă fiecărui element *x* din mulţimea A îi punem în corespondenţă un element *y* din mulţimea B şi numai unul singur, spunem că am definit o *funcţie* sau o *aplicaţie* pe mulţimea A cu valori în mulţimea B.

Evident, putem defini multe funcţii pe o mulţime A şi cu valori într-o mulţime B (B poate diferi de A sau să coincidă cu A). Pentru a le putea distinge, de obicei, fiecărei funcţii i se dă un nume. Faptul că *f* este o funcţie definită pe A şi cu valori în B, se notează astfel:

f:A -> B

Să observăm că notaţia, de mai sus, nu defineşte modul în care se realizează corespondenţa dintre elementele lui A şi B. Ea exprimă numai faptul că prin *f* s-a notat o astfel de corespondenţă.

Fie A o mulţime nevidă. Prin *lege de compoziţie* definită pe A se înţelege o aplicaţie *f* definită pe produsul cartezian AxA şi cu valori în A:

f:AxA -> A

Exemplu:

Cu N notăm mulţimea numerelor naturale {0,1,2,...}.

Produsul cartezian NxN este mulţimea perechilor de numere naturale:

(m,n)

Un exemplu de lege de compoziţie definită pe mulţimea numerelor naturale este operaţia de *adunare*, pe care o notăm cu simbolul "+":

+:NxN -> N

Operaţia de adunare pune în corespondenţă fiecărei perechi de numere naturale (m,n) un al treilea număr natural care se numeşte suma lor şi care se determină în mod unic.

O altă lege de compoziţie definită pe mulţimea numerelor naturale este operaţia de înmulţire, pe care o notăm cu simbolul "*":

*:NxN -> N

O mulţime se spune că formează un *grupoid* dacă este definită o lege de compoziţie internă pe mulţimea respectivă.

Mulţimea numerelor naturale este *grupoid* atît faţă de operaţia de adunare cît şi faţă de cea de înmulţire.

Legile de compoziţie au, de obicei, diferite proprietăţi. Conceptul de *grupoid* este cel mai general. El se află în vîrful ierarhiei. Se obţin concepte mai particulare în funcţie de proprietăţile legilor de compoziţie.

O proprietate a legilor de compoziţie întîlnită frecvent este cea de *asociativitate*. Aceasta se defineşte ca mai jos.

Fie A o mulţime pe care este definită operaţia notată cu "#":

#:AxA -> A

(A este grupoid faţă de operaţia notată cu "#").

Legea de compoziţie # este *asociativă* dacă:

(a#b)#c = a#(b#c)

oricare ar fi elementele *a, b, c* din A.

Un grupoid formează un *semigrup* dacă legea de compoziţie este asociativă.

Menţionăm că un grupoid faţă de operaţia de adunare se numeşte *grupoid aditiv*, iar un grupoid faţă de operaţia de înmulţire se numeşte *grupoid multiplicativ*.

În mod analog, un *semigrup* care este un grupoid aditiv este un *semigrup aditiv*, iar un semigrup care este un grupoid multiplicativ este un *semigrup multiplicativ*.

Mulţimea numerelor naturale este atît un semigrup aditiv cît şi un semigrup multiplicativ, deoarece ambele operaţii sînt asociative.

În general, pot să existe elemente care să aibă un comportament aparte faţă de legile de compoziţie. Astfel de elemente sînt, *elementele neutru*. Dacă notăm cu *e* un element neutru faţă de o lege de compoziţie # definită pe A, atunci:

e#x = x#e = x

unde:

x - Este un element oarecare al mulţimii A.

Nu orice semigrup are un element neutru. Semigrupul aditiv şi multiplicativ, ambele definite pe mulţimea numerelor naturale, au fiecare un element neutru. Astfel, zero este element neutru pentru semigrupul aditiv, iar unu este element neutru pentru semigrupul multiplicativ.

Se demonstrează că dacă există un element neutru faţă de o lege de compoziţie, atunci el este unic.

Să observăm că mulţimea:

M = {1,2,...} = N-{0}

este un semigrup aditiv care nu are element neutru. În mod analog,

P = {0,2,3,...} = N-{1}

este un semigrup multiplicativ care nu are element neutru.

Dacă un semigrup are o lege de compoziţie care are element neutru, atunci semigrupul respectiv se spune că formează un *monoid*. Noţiunile de *monoid aditiv* şi *monoid multiplicativ* rezultă imediat din cele de semigrup aditiv şi respectiv multiplicativ. Mulţimea N a numerelor naturale formează un monoid atît faţă de adunare, cît şi faţă de înmulţire. În schimb, mulţimea M = N-{0} nu este un monoid aditiv, iar P = N- {1} nu este un monoid multiplicativ.

Noţiunea de monoid a apărut ca o particularizare a noţiunii de semigrup.

O altă noţiune importantă în legătură cu legile de compoziţie este noţiunea de element *simetrizabil*.

Fie mulţimea A care este un monoid faţă de legea de compoziţie #, cu elementul neutru *e*.

Fie *x* un element al mulţimii A. Un element *y* din A este *simetricul* lui *x*, dacă:

x#y = y#x = e

Simetricul unui element, dacă există, este unic.

Un element se spune că este simetrizabil dacă are simetric.

Dacă orice element al unui monoid este simetrizabil, atunci monoidul respectiv este un *grup*.

Mulţimea N nu este grup nici faţă de operaţia de adunare şi nici faţă de operaţia de înmulţire. Într-adevăr, singurul element simetrizabil din mulţimea N, faţă de adunare, este 0, deoarece numai el are proprietatea:

0+0 = 0

În mod analog, singurul element simetrizabil din mulţimea N faţă de înmulţire este 1:

1*1 = 1

Mulţimea numerelor întregi, notată cu Z, formează un grup aditiv. Într-adevăr, adunarea numerelor întregi are toate proprietăţile necesare pentru a forma un grup:

- adunarea este asociativă, deci formează un semigrup;
- zero este element neutru faţă de adunare, deci este un monoid;
- dacă *x* este un întreg, atunci *-x* este simetricul lui, deoarece:

  x+(-x)=0

Mulţimea Z formează un monoid multiplicativ, dar nu şi un grup multiplicativ, deoarece numai unu şi -1 sînt simetrizabile faţă de înmulţire.

Legea de compoziţie # definită pe mulţimea A este *comutativă* dacă:

x#y = y#x

oricare ar fi elementele *x* şi *y* din mulţimea A.

Un grup se numeşte *comutativ* dacă legea de compoziţie definită pe mulţimea A, faţă de care A formează grup, este comutativă.

Mulţimea Z formează un grup aditiv comutativ deoarece adunarea numerelor întregi este comutativă.

O mulţime A este un *inel* dacă pe ea sînt definite două legi de compoziţie care au următoarele proprietăţi:

- A este grup comutativ faţă de una dintre legile de compoziţie;
- A este monoid faţă de cealaltă lege de compoziţie;
- legea de compoziţie faţă de care A este monoid, este distributivă faţă de cealaltă lege de compoziţie.

De obicei, legea de compoziţie faţă de care A este un grup comutativ se notează cu +, adică se consideră că grupul este aditiv. Cealaltă lege se obişnuieşte să se noteze cu *, adică se consideră că monoidul este multiplicativ.

Cu aceste notaţii, proprietatea de distributivitate se exprimă astfel:

$x*(y+z) = x*y+x*z$

oricare ar fi *x*, *y* şi *z* elemente ale lui A.

Mulţimile Z şi Q (mulţimea numerelor raţionale) sînt exemple de inele.

Un *inel* este *comutativ* dacă operaţia de înmulţire (*) este comutativă, adică:

$x*y = y*x$

oricare ar fi elementele *x* şi *y* ale lui A.

Inelele Z şi Q sînt comutative.

Un exemplu de inel necomutativ este mulţimea matricelor pătratice de ordinul *n* ($n > 1$) ale căror elemente sînt numere aparţinînd lui Z, Q sau R (mulţimea numerelor reale).

Elementul neutru al unui monoid aditiv se notează, de obicei cu zero, iar al unui monoid multiplicativ cu unu.

Un inel este *corp*, dacă elementele neutru ale celor două legi de compoziţie sînt diferite şi dacă orice element diferit de zero (elementul neutru al grupului aditiv) este simetrizabil în raport cu operaţia de înmulţire.

Ca exemplu de corpuri amintim mulţimile Q, R şi C (mulţimea numerelor complexe).

În fine, dacă inelul care este un corp, este comutativ, atunci corpul respectiv este şi el comutativ.

Din exemplele de mai sus, rezultă că ierarhizarea conceptelor se realizează din aproape în aproape. La început se stabilesc elementele

comune şi ele definesc conceptul cel mai general. Apoi, se fac particularizări adăugîndu-se proprietăţi noi şi se ajunge din aproape în aproape la concepte din ce în ce mai particulare (mai specializate).

Caracteristica de bază a unei ierarhii este proprietatea de moştenire.

Conform acestei proprietăţi, un concept al ierarhiei de pe un nivel are toate proprietăţile conceptului aflat pe nivelul imediat superior. De exemplu, dreptunghiul are toate proprietăţile paralelogramului, iar pătratul are toate proprietăţile dreptunghiului.

Ierarhiile pot apare la încercarea de a defini conceptele necesare pentru a rezolva o anumită problemă la un calculator.

Am văzut că pentru conceptele care nu corespund tipurilor predefinite din limbajul C++, putem defini clase prin care să se implementeze. Pînă în acest moment, clasele au fost definite considerîndu-se independente conceptele pe care le implementează. Este necesar ca ierarhia conceptelor să fie reflectată de către clasele care le implementează. Aceasta implică o noţiune nouă pentru a defini în primul rînd, proprietatea de bază a ierarhiilor *-moştenirea*. Proprietatea de moştenire se realizează cu ajutorul *claselor derivate*. Noţiunea de clasă derivată se defineşte ca mai jos.

Fie *c1* şi *c2* două concepte ale unei ierarhii aşa încît *c1* se află pe un nivel imediat superior faţă de conceptul *c2*. Cu alte cuvinte, conceptul *c2* se obţine din *c1* adăugînd caracteristici noi.

Fie *cl1* şi *cl2* clasele care implementează conceptele *c1* şi respectiv *c2*. Atunci clasa *cl2* se spune că este o clasă *derivată* a clasei *cl1* Aceasta înseamnă că clasa *cl2* are ca elemente membru toate elementele membru (date membru şi funcţii membru) ale clasei *cl1* (acestea sînt elemente moştenite), precum şi alte elemente specifice ei. Se obişnuieşte să se spună că clasa *cl2* moşteneşte elementele clasei *cl1*. Despre clasa *cl1* se spune că este o clasă de bază pentru clasa *cl2*.

Reluînd ierarhia care are în vîrf conceptul de *paralelogram*, vom presupune că, conceptele din ierarhie se implementează prin clase cu aceleaşi denumiri. Astfel, conceptul de *paralelogram* se implementează cu ajutorul clasei *paralelogram*, conceptul de *dreptunghi* cu ajutorul clasei *dreptunghi* şi aşa mai departe.

În acest caz, clasa *dreptunghi* este o clasă derivată a clasei *paralelogram*, iar clasa *paralelogram* este o clasă de bază a clasei *dreptunghi*. În mod analog, clasa *romb* este o clasă derivată a clasei *paralelogram*, iar clasa *paralelogram* este o clasă de bază a clasei *romb*.

În sfîrşit, clasa *patrat* este o clasă derivată a clasei *dreptunghi*, iar clasa *dreptunghi* este o clasă de bază a clasei *patrat*. În felul acesta, o ierarhie de

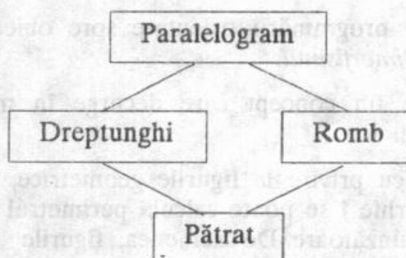
concepte conduce la o *ierarhie între clasele* care implementează conceptele ierarhiei respective. Clasa derivată se află totdeauna pe un nivel imediat inferior celui corespunzător clasei de bază. La baza ierarhiei dintre clase se află aceeaşi proprietate ca în cazul ierarhiei de concepte, adică proprietatea de moştenire.

O ierarhie stabilită între mai multe concepte poate să nu fie unică.

De exemplu, ierarhia care are în vîrf conceptul de *paralelogram* poate fi modificată aşa încît patratul să nu fie un caz particular al dreptunghiului, ci al rombului. Într- adevăr, pătratul se poate defini ca un romb care are un unghi drept. În acest caz, clasa *patrat* devine o clasă derivată a clasei *romb*, iar aceasta la rîndul ei devine o clasă de bază a clasei *patrat*.

Aşa cum s-a spus mai sus, un concept poate moşteni atribute de la mai multe concepte. Aceasta înseamnă că o clasă poate deriva din mai multe clase de bază. În acest caz, se spune că moştenirea este *multiplă* (clasa derivată moşteneşte de la mai multe clase de bază).

Revenind la exemplul de mai sus, să observăm că, conceptul de pătrat, se poate defini ca un dreptunghi care este în acelaşi timp şi un romb sau invers, adică un romb care este şi dreptunghi. Se obţine ierarhia din figura de mai jos.



În felul acesta, clasa *patrat* este o clasă derivată din clasele *dreptunghi* şi *romb*.

Programarea care permite definirea de ierarhii între tipurile abstracte de date pe baza conceptului de moştenire, se numeşte *programare orientată spre obiecte*.

În limbajul C++, ierarhia dintre tipurile abstracte de date se exprimă prin ierarhia de clase care implementează tipurile abstracte respective. Ierarhia dintre clase se stabileşte cu ajutorul noţiunilor de *clasă derivată* şi *clasă de bază*.

B. Stroustrup arată în lucrarea [14] că în cazul în care nu există ierarhii

între tipurile abstracte de date utilizate într- un program (tipuri abstracte independente), atunci este suficientă programarea prin abstractizarea datelor.

Programarea orientată spre obiecte, avînd la bază procesul de moştenire, este un salt calitativ faţă de programarea prin abstractizarea datelor.

Într-adevăr, procesul de moştenire implică un stil nou în programare.

În primul rînd, tipurile abstracte care au părţi comune nu se mai implementează independent. Fiecare tip abstract se implementează moştenind elementele comune de la alte tipuri abstracte aflate pe un nivel superior în ierarhia din care acesta face parte. În felul acesta, un tip abstract care are elemente comune cu altele se implementează adăugînd elemente noi la cele existente. Astfel de adăugări sînt posibile oricînd dacă se constată că este nevoie de noi tipuri abstracte de date. Aceasta înseamnă că, atunci cînd se constată că un program trebuie modificat sau dezvoltat, pe baza utilizării de noi tipuri abstracte de date, tipurile vechi, de obicei, nu se modifică, ci li se adaugă elemente noi. În felul acesta, problema reutilizării componentelor existente nu înseamnă modificarea sau reprogramarea lor, ci completarea lor cu elemente noi.

Programarea orientată spre obiecte utilizează şi alte concepte decît moştenirea.

Un alt concept al programării orientate spre obiecte, suportat de limbajul C++, este *polimorfismul.*

Polimorfismul este un concept care decurge în mod natural din conceptul de moştenire.

Reluînd exemplul cu privire la figurile geometrice, vom observa că fiecărei figuri din ierarhie i se poate calcula perimetrul şi aria, definind funcţii membru corespunzătoare. De asemenea, figurile ierarhiei pot fi afişate pe ecranul display-ului setat în mod grafic, pot fi deplasate etc. În acest scop este necesar să definim funcţii membru specifice pentru fiecare figură în parte. Astfel, de exemplu, este nevoie de o funcţie membru pentru clasa *paralelogram* care să afişeze paralelograme, o altă funcţie membru pentru clasa *dreptunghi* care să afişeze dreptunghiuri şi aşa mai departe. Aceste funcţii pot fi considerate ca fiind versiuni ale aceleeaşi funcţii. Aceasta este în concordanţă cu denumirea de polimorfism care vine din greacă şi înseamnă " a avea mai multe forme".

Problema care se pune în legătură cu funcţiile de acest fel este cea legată de selectarea versiunilor unei astfel de funcţii. Pentru a lămuri acest lucru să presupunem că în cazul ierarhiei de figuri geometrice am defini, pentru fiecare clasă, funcţia membru *afiseaza*, care afişează pe display obiectul

curent.

Astfel, dacă avem instanţierile:

```
paralelogram p(...);
dreptunghi d(...);
patrat pt(...);
romb r(...);
```

atunci un apel de forma:

```
p.afiseaza()
```

afişează paralelogramul definit de obiectul *p*. De asemenea:

```
d.afiseaza()  - afişează un dreptunghi;
pt.afiseaza() - afişează un pătrat;
r.afiseaza()  - afişează un romb.
```

La aceste apeluri se utilizează, de fiecare dată, funcţia membru *afiseaza* a clasei corespunzătoare obiectului prezent în apel.

În mod analog, se poate defini o funcţie *sterge* pentru fiecare din clasele ierarhiei care să anuleze afişarea obiectului curent.

Funcţiile *afiseaza* şi *sterge* se pot utiliza pentru a face deplasări pe ecran ale figurilor respective. Numim *deplasare* o funcţie care permite deplasarea unei figuri din poziţia curentă într-o poziţie nouă a ecranului. O astfel de funcţie poate fi definită cu doi parametri, *nx* şi *ny*, care definesc modificările coordonatelor vîrfurilor figurii care se deplasează. Numim *modif* funcţia care determină datele membru ale unei figuri în urma deplasării figurii respective. Evident, fiecare clasă a ierarhiei are o funcţie membru *modif* proprie (specifică).

Funcţia *deplasare* se poate defini ca mai jos:

```
void paralelogram::deplasare(int nx,int ny)
{
 sterge();    //sterge de pe ecran paralelogramul curent
 modif(nx,ny);  /* modifica coordonatele virfurilor
                   paralelogramului curent cu valorile nx si ny */
 afiseaza();    /* afiseaza paralelogramul in pozitia noua */
}
```

Funcţiile *sterge, modif* şi *afiseaza*, apelate în corpul funcţiei *paralelogram* sînt funcţii membru ale clasei *paralelogram*.

În mod analog, definim o funcţie membru *deplasare* pentru clasa *dreptunghi*:

```
void dreptunghi::deplasare(int nx,int ny)
{
 sterge();     //sterge de pe ecran dreptunghiul curent
 modif(nx,ny); /* modifica coordonatele virfurilor
                  dreptunghiului curent cu valorile nx si ny */
 afiseaza();   //afiseaza dreptunghiul in pozitia noua
}
```

În acest caz se apelează funcţiile membru *sterg*, *modif* şi *afiseaza* ale clasei *dreptunghi*.

Pentru celelalte clase (*patrat* şi *romb*) se va defini la fel funcţia *deplasare*.

Se observă că funcţia *deplasare* diferă de la o clasă la alta numai prin antet şi prin aceea că funcţiile pe care le apelează sînt funcţii membru ale aceleeaşi clase ca şi funcţia *deplasare*.

Se pune problema dacă nu se pot înlocui aceste funcţii cu una singură şi anume cu funcţia membru a clasei *paralelogram* care se află în vîrful ierarhiei.

În principiu, ea se poate aplica pentru obiecte ale claselor derivate, deoarece un astfel de obiect este totdeauna un *paralelogram* (orice dreptunghi, pătrat sau romb este un paralelogram).

O funcţie membru a unei clase de bază se poate aplica la obiecte ale claselor derivate deoarece acestea o moştenesc.

Problema apare la apelurile funcţiilor *sterge*, *modif* şi *afiseaza*, care aşa cum s-a afirmat mai sus, sînt funcţii membru ale clasei *paralelogram*. Aceste funcţii nu se pot utiliza dacă funcţia *deplasare* este apelată pentru un alt obiect decît cel al clasei *paralelogram*, deoarece ele diferă de la o figură geometrică la alta.

Funcţiile membru *sterge*, *modif* şi *afiseaza* se selectează la compilare, cînd se compilează fiecare funcţie membru *deplasare*. De aceea, deşi la execuţie, funcţia membru *deplasare* a clasei *paralelogram* se poate apela pentru un obiect al oricărei clase a ierarhiei, ea se execută corect numai pentru obiectele clasei *paralelogram*. În felul acesta, se pare că moştenirea funcţiei membru *deplasare* a clasei *paralelogram* pentru clasele derivate din clasa *paralelogram* devine un inconvenient şi nu permite deplasarea dreptunghiurilor, a pătratelor şi a romburilor.

Problema se poate rezolva dacă selectarea funcţiilor care să se apeleze în corpul funcţiei *deplasare* (*sterge*, *modif* şi *afiseaza*) nu s-ar face la compilarea funcţiei *deplasare*, ci la execuţie, la fiecare apel al funcţiei *deplasare*. În acest scop, s-a introdus noţiunea de funcţie *virtuală*.

O funcţie virtuală este o funcţie membru a unei clase a cărei prototip şi antet începe prin cuvîntul *virtual*. În cazul exemplului de mai sus, funcţiile *sterge, modif* şi *afiseaza* se vor defini ca funcţii membru virtuale pentru fiecare clasă din ierarhie.

Ca urmare a acestui fapt, apelul unei astfel de funcţii nu este rezolvat la compilare, ci la execuţie, selectîndu-se versiunea funcţiei respective care corespunde obiectului curent. Se obişnuieşte să se spună că apelul unei astfel de funcţii se realizează folosind *legătura întîrziată (late binding)* de obiect. Legăturile obişnuite dintre obiect şi apelul unei funcţii membru se rezolvă la compilare şi despre ele se spune că sînt *legături timpurii*. Ele se mai numesc şi *legături statice*, iar cele întîrziate se mai numesc şi *legături dinamice*.

În concluzie, definind funcţiile *sterge, modif* şi *afiseaza* ca funcţii virtuale, vom defini o singură funcţie membru *deplasare* pentru clasa *paralelogram*. Ea se poate aplica pentru un obiect al oricărei clase din ierarhie. La execuţia funcţiei *deplasare* se vor apela acele funcţii *sterge, modif* şi *afiseaza* care sînt funcţii membru *ale acelei clase* la care aparţine obiectul curent.

În încheierea acestui capitol amintim că programarea orientată spre obiecte admite şi alte concepte decît moştenirea şi polimorfismul. Astfel, un concept prezent în limbajele anterioare limbajului C++ şi care este suportat de programarea orientată spre obiecte este *identitatea obiectuală* (un exemplu de astfel de limbaj este smalltalk).

Alte concepte, cum este de exemplu *persistenţa obiectelor*, sînt suportate de extensii ale limbajului C++. Astfel de extensii sînt limbajele E şi O (vezi [6]).

29. CLASE DERIVATE ŞI CLASE DE BAZĂ

În capitolul 28 s-a arătat că între diferite concepte utilizate în programare se pot face ierarhizări. La baza ierarhizărilor se află procesul de moştenire, conform căruia un concept, diferit de cel din vîrful ierarhiei, are toate proprietăţile conceptelor care îl preced în ierarhia respectivă.

Ierarhia dintre concepte induce o ierarhie între clasele care implementează conceptele respective. În cazul claselor, procesul de moştenire înseamnă că o clasă diferită de cea din vîrful ierarhiei are ca elemente membru, elementele membru ale claselor care o preced, precum şi eventuale alte elemente membru specifice clasei respective. Elementele membru ale clasei aflate în vîrful ierarhiei sînt elemente comune pentru toate clasele ierarhiei respective.

Ierarhizarea claselor se face cu ajutorul noţiunii de *clasă derivată.*

Fie *cl1* şi *cl2* două clase ale unei ierarhii de clase, *cl2* fiind situată pe un nivel imediat inferior nivelului clasei *cl1*. Atunci clasa *cl2* se spune că este o *clasă derivată* a clasei *cl1*, iar *cl1* este o *clasă de bază* a clasei *cl2*. În acest caz, vom spune că clasa *cl2* moşteneşte clasa *cl1*.

Relaţia de derivare între două clase se exprimă la definirea clasei derivate. În acest scop, după numele clasei derivate, se scrie lista claselor de bază pentru clasa respectivă folosind ca separator două puncte. În felul acesta, o clasă derivată se defineşte ca mai jos:

```
class nume_clasa_derivata:
   lista_claselor_de_baza {
// elemente specifice clasei derivate
};
```

Lista aflată după două puncte indică clasele de bază pentru clasa derivată care se defineşte. Lista respectivă conţine, pentru fiecare clasă de bază, numele clasei care eventual este precedat de un modificator de protecţie. Ca modificatori de protecţie se pot folosi modificatorii *public* şi *private*. Ca separator între clasele de bază se foloseşte virgula.

Exemple:

1.

```
class cl2:cl1 {
...
```

```
};
```

Clasa *cl2* este o clasă derivată a clasei *cl1*. Clasa *cl1* este singura clasă de bază a clasei *cl2*.

2.

```
class cl2:private cl1 {
...
};
```

Clasa *cl2* este o clasă derivată a clasei *cl1*. Clasa *cl1* este singura clasă de bază a clasei *cl2*.

3.

```
class cl:cl1,public:cl2 {
...
};
```

Clasa *cl* este o clasă derivată a claselor *cl1* şi *cl2*.

Relaţia de derivare se poate defini şi pentru structuri. Aceasta înseamnă că exemplele de mai sus rămîn valabile şi în cazul în care cuvîntul *class* se schimbă cu *struct*. Deci, o clasă de tip *struct* poate fi atît o clasă derivată, cît şi o clasă de bază. Nu acelaşi lucru se poate afirma despre o clasă de tip *union*. O clasă de tip *union* nu poate fi nici clasă derivată şi nici clasă de bază.

În mod implicit, modificatorul de protecţie este *private* la definirea unei clase derivate şi *public* la definirea unei structuri derivate. De aici rezultă că exemplele 1 şi 2, de mai sus, sînt identice deoarece modificatorul *private* este implicit pentru clase.

Modificatorii de protecţie utilizaţi în *lista_claselor_de_bază* definesc protecţia în clasa derivată a elementelor moştenite.

Tabelul de mai jos indică accesul în clasa derivată a elementelor moştenite în funcţie de protecţia fiecărui element moştenit şi de modificatorul de protecţie utilizat în *lista_claselor_de_bază*.

Accesul în clasa de bază	**Modificatorul de protecţie din** *lista_claselor_de_baza*	**Accesul în clasa derivată a elementului moştenit**
private	private	inaccesibil
protected	private	private
public	private	private

(continuare)

Accesul în clasa de bază	Modificatorul de protecţie din *lista_claselor_de_baza*	Accesul în clasa derivată a elementului moştenit
private protected public	public public public	inaccesibil protected public

Din acest tabel se observă că o clasă derivată nu are acces la elementele clasei de bază care au protecţia *private*. În schimb, clasa derivată are acces la elementele clasei de bază ce au protecţia *protected* sau *public*.

Dacă la definirea clasei derivate se utilizează modificatorul de protecţie *private*, atunci elementele protejate prin *protected* sau *public* devin protejate prin protecţia *private*. Aceasta înseamnă că ele nu mai pot fi accesate de către o clasă derivată a clasei derivate în acest fel. De aceea, modificatorul *private* se va folosi în *lista_claselor_de_bază* numai dacă clasa derivată nu se utilizează ca o clasă de bază pentru o nouă derivare.

De obicei, o ierarhie de clase nu este o ierarhie finală, ea putînd fi dezvoltată adăugînd clase noi, care derivă din clasele terminale. Aceasta este posibil dacă la definirea claselor derivate se utilizează, în *lista_claselor_de_bază*, modificatorul *public* în loc de *private*. În acest caz, elementele protejate prin *protected* şi *public* se moştenesc în clasa derivată prin aceeaşi protecţie. De aceea, modificatorul de protecţie cel mai des utilizat în *lista_claselor_de_bază* este *public*.

Relaţia dintre clasa derivată şi clasele ei de bază ridică unele probleme pe care le vom analiza în continuare.

O primă problemă este aceea a relaţiilor dintre *constructorii* şi *destructorii* clasei derivate şi a claselor de bază. O a doua problemă este legată de *conversiile* obiectelor claselor derivate spre cele ale claselor de bază, precum şi a pointerilor spre astfel de obiecte.

În sfîrşit, o a treia problemă este legată de *redeclararea* datelor membru şi *supraîncărcarea* funcţiilor membru ale claselor de bază în clasa derivată.

29.1. Relaţia dintre constructorii şi destructorii claselor de bază şi ai clasei derivate

Constructorii şi destructorii sînt funcţii membru care nu se moştenesc.

La instanţierea unui obiect al unei clase derivate se apelează atît constructorii clasei derivate cît şi cei ai claselor de bază. Întîi se apelează constructorii claselor de bază şi la urmă se apelează şi constructorul clasei derivate. Ordinea de apel a constructorilor claselor de bază corespunde cu ordinea în care sînt indicate clasele respective în *lista_claselor_de_bază* din definiţia clasei derivate.

La distrugerea unui obiect al clasei derivate, destructorii se apelează în ordine inversă. Deci, întîi se apelează destructorul clasei derivate şi apoi se apelează destructorii claselor de bază în ordinea inversă faţă de ordinea de apel a constructorilor la instanţierea obiectului respectiv.

La instanţierea unui obiect se transmit valori parametrilor constructorilor pentru iniţializarea datelor membru. În cazul unui obiect al unei clase derivate, o parte din valori se folosesc pentru iniţializarea datelor membru specifice ale clasei derivate, iar restul pentru iniţializarea datelor membru ale claselor de bază.

Constructorul clasei derivate conţine parametri pentru toate valorile care se utilizează la iniţializarea obiectelor.

Transferul valorilor de iniţializare pentru datele membru ale claselor de bază se defineşte cu ajutorul antetului constructorului clasei derivate.

Fie A o clasă derivată din clasele de bază B1,B2,...,Bn:

```
class A:public B1,public B2,...,public Bn {
 ...
 A(...); //prototip pentru constructor
 ...
};
```

Constructorul A are un antet de forma:

```
A::A(...):B1(...),B2(...),...,Bn(...)
```

Constructorul A are o listă de parametri completă, adică pentru iniţializarea tuturor datelor membru ale unui obiect de tip A.

Expresia Bi(...) (i = 1,2,...,n) din antetul constructorului A conţine, în paranteză, o listă de expresii care definesc valori iniţiale pentru datele membru ale clasei Bi.

Dacă o clasă Bi nu are constructor, atunci pentru clasa respectivă nu va fi prezentă o expresie de forma Bi(...) în antetul constructorului A. De asemenea, expresia respectivă nu va fi prezentă în antetul lui A dacă datele membru ale clasei Bi se iniţializează cu valori implicite sau clasa Bi nu are date membru.

Expresia Bi(...) este un apel explicit al unui constructor al clasei Bi.

Ordinea în care se scriu expresiile Bi(...) în antetul constructorului A este arbitrară. Constructorii se apelează totdeauna în ordinea în care sînt scrise clasele în *lista_claselor_de_bază.*

O condiţie esenţială pentru a se putea apela constructorii claselor de bază prin intermediul clasei derivate este ca aceştia să aibă protecţia de tip *protected* sau *public.*

Lista care defineşte apelurile constructorilor claselor de bază nu este prezentă în prototipurile constructorilor claselor derivate ci numai în antetele acestora.

În general, clasele au constructori definiţi de programatori. Totuşi există situaţii în care o clasă nu are constructori definiţi de programator. În acest caz, compilatorul generează în mod automat un constructor implicit (fără parametri) pentru clasa respectivă. Avînd în vedere această situaţie rezultă că teoretic pot apare următoarele cazuri:

1. Clasa derivată şi clasele de bază ale ei nu au constructori definiţi de programator.
2. Cel puţin o clasă dintre clasele de bază are constructori, iar clasa derivată nu are constructori.
3. Clasa derivată are constructori iar clasele de bază ale ei nu au constructori definiţi de programator.
4. Clasa derivată are constructori şi cel puţin o clasă de bază a ei are constructori.

În primul caz, la instanţierea obiectelor clasei derivate se aplică constructorii impliciţi generaţi de compilator.

În cazul al doilea, toţi constructorii claselor de bază definiţi de programator trebuie să fie impliciţi sau să aibă toţi parametri impliciţi. La instanţierea obiectelor clasei derivate nu se pot face iniţializări. La o astfel de instanţiere se aplică constructorul implicit al clasei derivate, generat de compilator şi constructorii claselor de bază definiţi de programator, folosindu-se valorile implicite ale parametrilor acestora.

În cazul al treilea, constructorul clasei derivate realizează toate iniţializările atît pentru datele membru specifice clasei derivate cît şi pentru datele membru ale claselor de bază. De aceea, în acest caz este necesar ca clasa derivată să aibă acces la toate datele membru ale claselor de bază care comportă iniţializări.

Constructorul clasei derivate are parametri corespunzători pentru datele

care comportă iniţializări. Valorile lor se atribuie datelor membru în corpul constructorului. Evident, antetul constructorului este format numai din numele lui urmat de lista parametrilor formali, inclusă în paranteze rotunde.

În cazul al patrulea, constructorul clasei derivate conţine parametri pentru toate datele membru care comportă iniţializări. Valorile parametrilor corespunzători datelor membru specifice clasei derivate se utilizează pentru a iniţializa datele respective în mod obişnuit (de obicei în corpul constructorului clasei derivate se iniţializează aceste date),iar celelalte valori se utilizează la apelul explicit al constructorilor claselor de bază.

Apelurile explicite ale constructorilor sînt situate în antetul constructorului clasei derivate, aşa cum s-a indicat mai sus.

Exemple:

1. Clasa A este o clasă derivată a clasei de bază B. Nici una din clase nu are constructor.

   Fie instanţierea:

   ```
   A a;
   ```

   Obiectul *a* se instanţiază fără a se face iniţializări. La instanţiere se aplică constructorii impliciţi ai claselor generaţi de compilator.

2. Clasa A este o clasă derivată a clasei de bază B.

   Clasa B are constructor cu toţi parametrii impliciţi:

   ```
   class B {
     protected:
       double x;
       double y;
     public:
       B(double p=0,double q=0)
       {
          x = p;
          y = q;
       }
       ...
   };
   ```

   Clasa A nu are constructor:

```
class A:public B{
  protected:
    Boolean ecran;
  public:
  ·  interior();
  ...
};
```

Fie declaraţia:

A a;

Obiectul *a* se instanţiază apelîndu-se constructorul cu parametri impliciţi al clasei B.

Datele membru *x* şi *y* se iniţializează cu valoarea 0, iar data membru *ecran* nu este iniţializată.

3. Clasa A este o clasă derivată a clasei B. Clasa B nu are constructor definit de utilizator şi se defineşte ca mai jos:

```
class B {
   protected:
      double x;
      double y;
   public:
   ...    // nu exista constructori
};
```

Clasa A are un constructor:

```
class A:public B {
   protected:
      Boolean ecran;
      int abs,ord;
   public:
      A(double p=0,double q=0,int lx=639,
        int ly=349)
      {
         // initializeaza datele membru ale clasei de baza
         x=p;
         y=q;

         // initializeaza datele membru specifice
         abs=lx;
         ord=ly;
```

```
        if(0 <= p && p <= lx && 0 <= q &&
            q <= ly)
            ecran = true;
        else
            ecran = false;
    }
    ...
};
```

Exemple de instanţieri:

```
A a; /* a.x=0, a.y=0, a.abs=639, a.ord=349, a.ecran=true */
A b(100,400); /* b.x=100, b.y=400, b.abs=639,
                 b.ord=349, b.ecran = false   */
```

4. Clasa A este o clasă derivată a clasei B şi ambele clase au constructori definiţi de programator.

```
class B {
   protected:
       double x,y;
   public:
       B(double xx=0,double yy=0)
       {
           x=xx;
           y=yy;
       }
       ...
};
class A:public B {
   protected:
       Boolean ecran;
       int abs,ord;
   public:
       A(double p=0,double q=0,int lx=639,
         int ly=349):B(p,q)
       {
           abs=lx;
           ord=ly;
           if(0 <= p && p <= lx && 0 <= q && q <=
           ly)
               ecran = true;
           else
               ecran = false;
       }
```

```
    ...
};
```

Se pot utiliza aceleaşi instanţieri ca în exemplul de la punctul 3. Se realizează aceleaşi iniţializări ca la punctul 3.

Amintim că datele *abs* şi *ord* pot fi iniţializate chiar în antetul constructorului, ca mai jos:

```
    A(double p=0,double q=0,int lx=639,
     int ly=349):B(p,q),abs(lx),ord(ly)
    {
       if(0 <= p && p <= lx && 0 <= q && q <=
       ly)
          ecran = true;
       else
          ecran = false;
    }
    ...
};
```

O clasă derivată este necesar să aibă cel puţin un constructor în cazul în care cel puţin o clasă de bază are un constructor care nu este implicit sau nu are toţi parametrii impliciţi.

Punctele 1 - 4 de mai sus, au fost analizate avîndu-se în vedere constructorii obişnuiţi, adică constructorii care nu sînt de copiere.

În lipsa constructorilor de copiere, compilatorul generează în mod automat constructori de copiere impliciţi care, atunci cînd se aplică, copiază datele membru ale obiectului sursă (care se copiază) în datele membru corespunzătoare ale obiectului destinaţie (care se crează). O copiere de acest fel a fost numită *copiere bit cu bit*.

Copierea bit cu bit este adesea suficientă şi atunci nu este nevoie să se definească constructori de copiere. Există însă cazuri cînd copierea bit cu bit nu este suficientă şi atunci este nevoie de astfel de constructori. Ei se utilizează atît la iniţializarea unui obiect printr-un alt obiect, cît şi la transferul prin parametri ai obiectelor şi la returnarea acestora la revenirea din funcţii.

Constructorii de copiere sînt necesari mai ales atunci cînd clasele au ca date membru pointeri spre zone alocate dinamic în memoria *heap*. De asemenea, în condiţiile moştenirii unor date membru ale claselor de bază, pot apare situaţii în care copierea bit cu bit poate conduce la erori.

În legătură cu apelul constructorilor de copiere vom distinge şi în acest caz mai multe situaţii.

Prima situaţie, cînd nici clasa derivată şi nici clasele de bază nu au constructori de copiere, a fost deja amintită mai sus şi anume, în acest caz, se face o copiere bit cu bit apelîndu-se constructorii de copiere impliciţi generaţi de compilator.

Presupunem acum situaţia în care clasa derivată nu are constructor de copiere, în schimb clasele de bază, o parte din ele sau toate, au constructori de copiere. Dacă este nevoie de un constructor de copiere pentru un obiect al clasei derivate (iniţializare prin copiere, transfer de obiecte prin parametri sau obiect returnat de o funcţie), atunci se apelează constructorul de copiere implicit al clasei derivate. Acesta realizează o copiere bit cu bit a datelor specifice clasei derivate şi a datelor membru moştenite de la clasele de bază care nu au constructori de copiere, iar celelalte date membru se iniţializează apelîndu-se automat constructorii de copiere ai claselor de bază corespunzătoare.

O altă situaţie este aceea în care clasa derivată are constructor de copiere. În acest caz, se apelează constructorul de copiere al clasei derivate, cînd este nevoie de un astfel de constructor (vezi mai sus), iar utilizarea constructorilor claselor de bază este în funcţie de definiţia constructorului de copiere al clasei derivate.

O primă regulă este aceea că, apelul constructorului de copiere al clasei derivate nu conduce la apelul automat al constructorilor de copiere ai claselor de bază.

Fie A o clasă derivată şi B o clasă de bază a ei. Să presupunem că A are un constructor de copiere de antet:

```
A(const A& a)
```

În acest caz, clasa de bază B, fie că nu are nici un constructor definit de programator, fie că are un constructor definit de programator care nu are parametri sau are numai parametri impliciţi.

La o instanţiere de forma:

```
A b=a;
```

unde *a* este un obiect al clasei *A* în prealabil declarat, se apelează constructorul de copiere al clasei A cu antetul indicat mai sus. Apelul constructorului de copiere al clasei A conduce în mod automat la apelul constructorului implicit al clasei B (cel generat de compilator sau cel definit de programator dacă există un astfel de constructor) sau, în caz contrar, la apelul constructorului clasei B care are toţi parametrii impliciţi.

Dacă clasa de bază B are constructori dar nici unul nu este implicit sau cu toţi parametri impliciţi, atunci constructorul de copiere al clasei derivate

A trebuie să conţină apelul explicit al unui constructor al clasei B. Cu alte cuvinte, constructorul de copiere al clasei A va avea antetul:

A(const A& a):B(...)

Exerciţii:

29.1 Să se implementeze tipul abstract *punct* pentru prelucrarea punctelor din plan.

Clasa *punct* are două date membru care definesc poziţia punctelor în plan:

x - abscisa

şi

y - ordonata

Protecţia acestor date este de tip *protected* pentru a putea fi moştenite prin derivare.

Clasa are un constructor cu parametrii impliciţi, valorile implicite fiind zero.

Funcţiile membru ale clasei *punct* sînt:

- *citpct* - citeşte coordonatele unui punct;
- *afispct* - afişează coordonatele unui punct;
- supraîncărcarea operatorilor +, - şi == pentru obiecte de tip punct.
- supraîncărcarea produsului unui obiect cu un număr;
- supraîncărcarea împărţirii unui obiect cu un număr;
- *dist* - distanţa dintre două puncte;
- *retabs* - returnează abscisa punctului curent;
- *retord* - returnează ordonata punctului curent.

FIŞIERUL BXXIX1.H

```
class punct {
 protected:
     double x0;  //abscisa
     double y0;  //ordonata
```

```
    static int rcit_double (const char *text,
                            double& d);
    /* - afiseaza text;
       - citeste un numar si-l pastreaza in zona referita de d;
       - returneaza:
            0 - la sfirsitul de fisier;
            1 - altfel.
    */

public:
    punct(double abs = 0,double ord=0); //constructor

    int citpct(const char *text);
    /* - afiseaza sirul spre care pointeaza text daca nu este vid;
       - citeste coordonatele obiectului curent;
       - returneaza:
            0 - la sfirsitul de fisier;
            1 - altfel.
    */

    void afispct(const char *f) const;
    /* afiseaza coordonatele obiectului curent folosind formatul f sau
       un format standard daca f pointeaza spre sirul vid */

    friend punct operator +(punct p1,punct p2);
    // returneaza p1+p2

    friend punct operator -(punct p1,punct p2);
    // returneaza p1-p2

    punct operator *(double d);
    // returneaza p*d; p este obiectul curent

    friend punct operator *(double d, punct p2);
    // returneaza d*p2

    punct operator /(double d);
    // returneaza p/d; p este obiectul curent

    double dist(punct p2) const;
    /* returneaza distanta dintre punctele p si p2;
       p este obiectul curent */
```

```
    double retabs() const;
    // returneaza abscisa obiectului curent

    double retord() const;
    // returneaza ordonata obiectului curent

    friend int operator == (punct p1,punct p2);
    // returneaza 1 daca p1 si p2 coincid si 0 altfel

    punct operator -();
    // returneaza simetricul fata de origine al
    // obiectului curent
};
```

Funcţiile membru se definesc în fişierul de extensie *CPP*, de mai jos.

FIŞIERUL BXXIX1

```
#ifndef __MATH_H
#include <math.h>
#define __MATH_H
#endif
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
#ifndef __PUNCT_H
#include "BXXIX1.H"
#define __PUNCT_H
#endif

int punct::rcit_double(const char *text,double& d)
/* - afiseaza sirul spre care pointeaza text;
   - citeste un numar si-l pastreaza in zona referita de d;
   - returneaza:
         0 - la sfirsit de fisier;
         1 - altfel.
*/
{
 char t[255];
 double x;

 for(;;) {
    printf(text);
    if(gets(t)==0)
```

```
        return 0;  //EOF
      if(sscanf(t,"%lf",&x)==1)
        break;
      printf("nu s-a tastat un numar\n");
  }
  d=x;
  return 1;
 }  //sfirsit rcit_double

inline punct::punct(double abs,double ord)
// constructor
{
 x0=abs;
 y0=ord;
}  //sfirsit constructor

int punct::citpct(const char *text)
/* - afiseaza sirul spre care pointeaza text daca nu este vid;
   - citeste coordonatele obiectului curent;
   - returneaza:
          0 - la sfirsit de fisier;
          1 - altfel.
*/
{
 if(text && *text)
     printf(text);
 if(rcit_double("\nabscisa=",x0)==0)
     return 0;
 if(rcit_double("\nordonata=",y0)==0)
     return 0;
 return 1;
}  //sfirsit citpct

void punct::afispct(const char *f) const
/* afiseaza coordonatele obiectului curent folosind formatul f sau un
   format standard daca f pointeaza spre sirul vid */
{
 if(f && *f)
     printf(f,x0,y0);
 else
     printf("x0=%g\y0=%g\n",x0,y0);
}  //sfirsit afispct
```

```
punct operator +(punct p1,punct p2)
// returneaza p1+p2
{
 punct r;
     r.x0 = p1.x0+p2.x0;
     r.y0 = p1.y0+p2.y0;
 return r;
}   // sfirsit operator +

punct punct::operator -()
// returneaza simetricul fata de origine al obiectului curent
{
 punct r;
     r.x0 = -x0;
     r.y0 = -y0;
 return r;
}  // sfirsit operator - unar

punct operator -(punct p1,punct p2)
// returneaza p1-p2
{
 return p1 + -p2;
} // sfirsit operator - binar

punct punct::operator *(double d)
// returneaza p*d; p este obiectul curent
{
 punct r;
     r.x0 = x0*d;
     r.y0 = y0*d;
 return r;
} // sfirsit operator * (double)

punct operator * (double d,punct p2)
// returneaza d*p2
{
 return p2*d;
} // sfirsit operator *(double,punct)

punct punct::operator /(double d)
// returneaza p/d; p este obiectul curent;
// returneaza originea axelor daca d=0
{
```

```
punct r;   //x0=y0=0
if(d==0)
    return r;
r.x0=x0/d;
r.y0=y0/d;
return r;
}   // sfirsit operator /

inline double punct::dist(punct p2) const
/* returneaza distanta dintre punctele p si p2;
   p este obiectul curent */
{
 return sqrt((x0-p2.x0)*(x0-p2.x0)+(y0-p2.y0)*
          (y0-p2.y0));
}   // sfirsit dist

inline double punct::retabs() const
// returneaza abscisa obiectului curent
{
 return x0;
} // sfirsit retabs

inline double punct::retord() const
// returneaza ordonata obiectului curent
{
 return y0;
} // sfirsit retord

inline int operator == (punct p1,punct p2)
/* returneaza:
    1 - daca p1 si p2 coincid;
    0 - altfel.
*/
{
 return p1.x0==p2.x0 && p1.y0==p2.y0;
} // sfirsit operator ==
```

29.2 Să se implementeze tipul abstract *gpunct* care pe lîngă elementele membru ale clasei *punct*, definită în exercițiul 29.1, conține elemente necesare pentru a afișa un punct în mod grafic pe ecran.

Clasa *gpunct* este o clasă derivată a clasei *punct* definită în exercițiul precedent.

Clasa *gpunct* are următoarele elemente specifice:

- date membru:
  - *culoare* - variabilă de tip *int* care defineşte culoarea pentru afişarea punctului (pentru detalii se va revedea capitolul 19);
  - *vizibil* - are valoarea 1 dacă punctul este afişat pe ecran şi zero altfel.
- funcţii membru:
  - *afiseaza* - afişează punctul în mod grafic pe ecran;
  - *sterge* - şterge punctul de pe ecran (culoarea lui devine aceeaşi cu a fondului);
  - constructor pentru iniţializarea elementelor membru ale unui punct; toţi parametri sînt impliciţi;
  - constructor cu doi parametri: un parametru este de tip *punct*, iar celălalt defineşte culoarea; acesta din urmă este implicit.

FIŞIERUL BXXIX2.H

```
#ifndef __PUNCT_H
#include "BXXIX1.CPP"
#define __PUNCT_H
#endif
#ifndef __GRAPHICS_H
#include <graphics.h>
#define __GRAPHICS_H
#endif

class gpunct:public punct {
 protected:
    int culoare;
    int vizibil;
 public:
    gpunct(double abs=0,double ord=0,int c=WHITE);
    /* constructor; in mod implicit punctul are coordonatele zero,
       culoarea este definita cu ajutorul constantei simbolice
       WHITE(alb) si nu este afisat la instantiere */

    gpunct(const punct& p,int c=WHITE);
    /* constructor utilizat la initializarea obiectului curent folosind un
       obiect de tip punct existent */
```

```
    void afiseaza();
    // afiseaza obiectul curent conform atributelor sale

    void sterge();
    // face punctul invizibil
};
```

Funcţiile membru ale clasei *gpunct* se definesc în fişierul de mai jos, de extensie *CPP*.

FIŞIERUL BXXIX2

```
#ifndef __GPUNCT_H
#include "BXXIX2.H"
#define __GPUNCT_H
#endif

inline gpunct::gpunct(double abs,double ord,int c):
                       punct(abs,ord)
// implicit x0=y0=0 si c=WHITE
{
 culoare=c;
 vizibil=0;   // la instantiere punctul nu este vizibil
} // sfirsit constructor

inline void gpunct::afiseaza()
// afiseaza punctul curent
{
 putpixel(x0,y0,culoare);
 vizibil=1;
} // sfirsit afiseaza

inline void gpunct::sterge()
// punctul devine invizibil
{
 // afiseaza punctul cu culoarea de fond curenta
 putpixel(x0,y0,getbkcolor());
 vizibil=0;
}

gpunct::gpunct(const punct& p,
              int c):punct(p.retabs(),p.retord())
// obiectul curent se initializeaza cu coordonatele obiectului p
{
 culoare = c;
```

```
 vizibil = 0;
}
```

29.3 Să se scrie un program care citeşte coordonatele a două puncte şi le afişează împreună cu mijlocul segmentului format de cele două puncte. Punctele citite se afişează colorate cu culoarea implicită, iar punctul din mijlocul segmentului se afişează cu culoarea definită de constanta simbolică LIGHTGREEN.

PROGRAMUL BXXIX3

```
#include <conio.h>
#include <stdlib.h>
#include "BXXIX2.CPP"

main()
/* - citeste coordonatele a doua puncte;
   - afiseaza punctele respective de culoare implicita impreuna cu
     punctul din mijlocul segmentului format de punctele citite.
     Punctul din mijlocul segmentului se afiseaza cu culoarea definita
     de constanta simbolica LIGHTGREEN. */
{
 gpunct a,b;

 if(a.citpct("punctul a")==0) {
    printf("s-a citit EOF\n");
    exit(1);
 }
 if(b.citpct("punctul b")==0){
    printf("s-a citit EOF\n");
    exit(1);
 }
 gpunct c((a+b)/2, LIGHTGREEN);
 int gd=DETECT,gm;
 initgraph(&gd,&gm,"c:\\borlandc\\bgi");
 a.afiseaza();
 b.afiseaza();
 c.afiseaza();
 getch();
 closegraph();
}
```

Observaţii:

1. Funcţia *citpct* se poate apela cu obiecte de tip *gpunct* deoarece ea este moştenită de la clasa *punct* care este clasa de bază pentru *gpunct*.

2. Supraîncărcările operatorilor pentru obiectele de tip *punct* se moştenesc pentru clasele derivate şi ele se pot aplica şi pentru obiecte derivate. Din această cauză, expresia (a+b)/2 este corectă, *a* şi *b* fiind instanţieri ale clasei *gpunct* care este o clasă derivată a clasei *punct*.

   Rezultatul expresiei de mai sus este un obiect de tip *punct*.

29.4 Să se implementeze tipul abstract *gtext* pentru afişarea de texte pe ecranul setat în mod grafic.

Un text se poate afişa în mod grafic folosind funcţia *outtextxy* din biblioteca standard a limbajelor C şi C++.

Ea este de prototip:

```
void far outtextxy(int x,int y,char far *sir);
```

unde:

x şi y - Definesc coordonatele de început ale cîmpului în care se afişează textul spre care pointează *sir*.

Pentru a afişa un text în mod grafic este necesar să definim anumite caracteristici şi anume:

- setul de caractere;
- dimensiunea caracterelor;
- cadrajul caracterelor;
- direcţia de scriere a caracterelor;
- culoarea caracterelor.

Aceste caracteristici, exceptînd culoarea, se definesc cu ajutorul a două funcţii: *settextstyle* şi *settextjustify* (pentru detalii se poate consulta paragraful 19.4).

Prima funcţie are prototipul:

```
void far settextstyle(int font,int direction,
                      int charsize);
```

Cea de a doua funcţie are prototipul:

```
void far settextjustify(int oriz,int vert);
```

Constructorul clasei *gtext* defineşte valorile pentru parametri acestor

funcţii.

Funcţiile respective se apelează folosind aceste valori înainte de a afişa textul cu ajutorul funcţiei *outtextxy*.

Întrucît după afişarea textului pe ecran este nevoie să se revină la caracteristicile vechi pentru afişarea textelor, înainte de a apela funcţiile *settextstyle* şi *settextjustify* se vor salva caracteristicile curente cu ajutorul funcţiei *gettextsettings* de prototip:

```
void far gettextsettings(struct textsettingstype
                                far *textinfo);
```

unde tipul utilizator *textsettingstype* se defineşte în fişierul *graphics.h* astfel:

```
struct textsettingstype {
        int font;
        int direction;
        int charsize;
        int horiz;
        int vert;
};
```

De aici rezultă că pentru a afişa un text se vor urma paşii:

1. Se memorează caracteristicile curente apelînd funcţia *gettextsettings*.
2. Se apelează funcţiile *settextstyle* şi *settextjustify* pentru a seta caracteristicile obiectului curent.
3. Se afişează textul apelînd funcţia *outtextxy*.
4. Se refac caracteristicile păstrate la punctul 1 prin apelul funcţiilor *settextstyle* şi *settextjustify*.

În mod implicit, se foloseşte culoarea definită de constanta simbolică WHITE pentru afişarea caracterelor textului.

Se pot afişa texte colorate cu alte culori indicînd o valoare corespunzătoare pentru parametrul *culoare* al constructorului. Pentru gestiunea culorilor se folosesc funcţiile *setcolor* şi *getcolor* (vezi paragraful 19.2).

FIŞIERUL BXXIX4.H

```
#ifndef __GRAPHICS_H
#include <graphics.h>
#define __GRAPHICS_H
#endif
```

```
class gtext {
 protected:
     int font;
     int directie;
     int dimensiune_caracter;
     int cadraj_oriz;
     int cadraj_vert;
     int culoare_text;
     char text[11];
     int vizibil_text;
 public:
     gtext();    // constructor implicit

     gtext(char *sir,
          int culoare = WHITE,
          int f = DEFAULT_FONT,
          int dir = HORIZ_DIR,
          int dim_car = 1,
          int cadraj_o = CENTER_TEXT,
          int cadraj_v = CENTER_TEXT);

     void afisgtext(int x,int y);
     /* afiseaza textul definit de obiectul curent intr-un cimp care
        incepe in punctul de coordonate (x,y) */

     void stergegtext(int x,int y);
     /* face textul invizibil; textul este afisat in cimpul care incepe
        in punctul de coordonate (x,y) */
};
```

Funcţiile membru ale clasei *gtext* se definesc în fişierul de mai jos, de extensie *CPP*.

FIŞIERUL BXXIX4

```
#ifndef __GTEXT_H
#include "BXXIX4.H"
#define __GTEXT_H
#endif
#ifndef __STRING_H
#include <string.h>
#define __STRING_H
#endif
```

```
gtext::gtext()
/* - constructor implicit;
   - toate datele membru au valori implicite;
   - textul contine sirul vid.
*/
{
 text[0] = '\0';
 culoare_text = WHITE;
 font = DEFAULT_FONT;
 directie = HORIZ_DIR;
 dimensiune_caracter = 1;
 cadraj_oriz = CENTER_TEXT;
 cadraj_vert = CENTER_TEXT;
 vizibil_text = 0;
}

gtext::gtext(char *sir,int culoare,int f,int dir,
             int dim_car,int cadraj_o,int cadraj_v)
/* constructor cu parametrii impliciti in afara de primul */
{
 strncpy(text,sir,11);  /* copiaza cel mult primele 10
                           caractere ale sirului spre
                           care pointeaza sir */
 text[10] = '\0';  /* se forteaza sfirsitul sirului de caractere */
 culoare_text = culoare;
 font = f;
 directie = dir;
 dimensiune_caracter = dim_car;
 cadraj_oriz = cadraj_o;
 cadraj_vert = cadraj_v;
 vizibil_text = 0;    // textul nu este afisat la instantiere
}

void gtext::afisgtext(int x,int y)
/* afiseaza textul definit de obiectul curent intr-un cimp care incepe in
   punctul de coordonate (x,y) */
{
 struct textsettingstype caract_crt;
 int culoare_crt;

 /* se salveaza caracteristicile curente */
 gettextsettings(&caract_crt);
 culoare_crt = getcolor();
```

```
  /* se seteaza caracteristicile obiectului */
  settextstyle(font,directie,dimensiune_caracter);
  settextjustify(cadraj_oriz,cadraj_vert);
  setcolor(culoare_text);

  /* afiseaza textul */
  outtextxy(x,y,text);

  /* se refac caracteristicile */
  settextstyle(caract_crt.font,caract_crt.direction,
               caract_crt.charsize);
  settextjustify(caract_crt.horiz,caract_crt.vert);
  setcolor(culoare_crt);
  vizibil_text = 1;   // textul este vizibil
}

void gtext::stergegtext(int x,int y)
/* face invizibil textul care este afisat in cimpul care incepe in punctul de
   coordonate (x,y) */
{
  int culoare_temp = culoare_text;
  // salveaza culoarea textului obiectului curent

  culoare_text = getbkcolor();
  // culoarea obiectului curent devine culoarea fondului

  afisgtext(x,y);    // textul devine invizibil
  culoare_text = culoare_temp;
  // se reface culoarea obiectului curent

  vizibil_text = 0;   // textul este invizibil
}
```

24.5 Să se implementeze tipul abstract *gtextpunct* care permite afişarea punctelor şi a textelor în mod grafic.

Tipul *gtextpunct* se defineşte printr-o clasă derivată care are două clase de bază: *gpunct* şi *gtext*. Clasa *gtextpunct* moşteneşte elementele membru ale ambelor clase.

FIŞIERUL BXXIX5.H

```
#ifndef __GTEXT_H
#include "BXXIX4.CPP"
```

```
#define __GTEXT_H
#endif
#ifndef __GPUNCT_H
#include "BXXIX2.CPP"
#define __GPUNCT_H
#endif

class gtextpunct:public gpunct,public gtext {
 protected:
    int vizibil_punct_text;
 public:
    gtextpunct();  //constructor implicit
    gtextpunct(char *sir,
               double x = 0,
               double y = 0,
               int c = WHITE,
               int culoare = WHITE,
               int f = DEFAULT_FONT,
               int dir = HORIZ_DIR,
               int dim_car = 1,
               int cadraj_o = CENTER_TEXT,
               int cadraj_v = CENTER_TEXT);

    void afisgtextpunct(int xt,int yt);
    /* afiseaza in mod grafic punctul si textul definit de obiectul
       curent */

    void stergegtextpunct(int xt,int yt);
    /* face invizibil punctul si textul definit de obiectul curent */
};
```

Funcţiile membru se definesc în fişierul de extensie *CPP*, de mai jos.

FIŞIERUL BXXIX5

```
#ifndef __GTEXTPUNCT_H
#include "BXXIX5.H"
#define __GTEXTPUNCT_H
#endif

inline gtextpunct::gtextpunct()
/* - constructor implicit; textul contine sirul vid, iar celelalte date
     membru au valori implicite;
   - se utilizeaza la instantierea obiectelor neinitializate. */
{
```

```
 // obiectul nu se afiseaza la instantiere
 vizibil_punct_text = 0;
}

inline gtextpunct::gtextpunct(char *sir,
       double x,double y,
       int c,int culoare,int f,int dir,
        int dim_car,int cadraj_o,
        int cadraj_v):
        gtext(sir,culoare,f,dir,dim_car,cadraj_o,
       cadraj_v),gpunct(x,y,c)
// constructor pentru initializarea obiectelor la instantiere
{
 vizibil_punct_text = 0;
}

inline void gtextpunct::afisgtextpunct(int xt,
                                        int yt)
/* - afiseaza punctul si textul obiectului curent;
   - textul se afiseaza intr-o zona care incepe in punctul de coordonate
     (xt,yt). */
{
 afiseaza();   // afiseaza punctul
 afisgtext(xt,yt);  // afiseaza textul
 vizibil_punct_text = 1;
}

inline void gtextpunct::stergegtextpunct(int xt,
                                          int yt)
/* face invizibil punctul si textul obiectului curent */
{
 sterge();    // face punctul invizibil
 stergegtext(xt,yt);   // face textul invizibil
 vizibil_punct_text = 0;
}
```

29.6 Să se modifice programul din exerciţiul 29.3 în aşa fel încît împreună cu afişarea punctelor să se afişeze şi literele care notează punctele respective.

Punctele se notează astfel:

A - primul punct citit de la intrarea standard;
B - al doilea punct citit de la intrarea standard;

M - punctul aflat la mijlocul segmentului AB.

Literele care notează punctele se afişează deplasate faţă de punctele respective după cum urmează:

A - abscisa deplasată în stînga cu 10 pixeli;

B - ordonata deplasată în jos cu 10 pixeli;

M - abscisa deplasată în dreapta cu 10 pixeli;
- ordonata deplasată în sus cu 10 pixeli.

PROGRAMUL BXXIX6

```
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#include "BXXIX5.CPP"

main()
/* - citeste 2 puncte de la intrarea standard;
   - afiseaza punctele respective si punctul situat la mijlocul segmentului
     determinat de ele impreuna cu literele:
         A pentru primul punct;
         B pentru al doilea punct;
         M pentru punctul situat la mijlocul segmentului AB;
   - culori:
         - punctele A si B se afiseaza folosind culoarea implicita;
         - punctul M se afiseaza colorat cu culoarea definita de constanta
           simbolica LIGHTGREEN;
         - caracterele A si B se afiseaza folosind culoarea implicita;
         - caracterul M se afiseaza colorat cu culoarea definita de
           constanta simbolica YELLOW.
*/
{
 gtextpunct a("A");
 gtextpunct b("B");

 if(a.citpct("punctul A") == 0) {
    printf("s-a citit EOF\n");
    exit(1);
 }
```

```
    if(b.citpct("punctul B") == 0) {
        printf("s-a citit EOF\n");
        exit(1);
    }
    gpunct c=(a+b)/2;
    gtextpunct
    m("M",c.retabs(),c.retord(),LIGHTGREEN,
                                YELLOW);
    int gd = DETECT,gm;
    initgraph(&gd,&gm,"c:\\borlandc\\bgi");
    a.afisgtextpunct(a.retabs()-10,a.retord());
    b.afisgtextpunct(b.retabs(),b.retord()+10);
    m.afisgtextpunct(m.retabs()+10,m.retord()-10);
    getch();
    closegraph();
}
```

29.2. Conversii

În capitolul 24 s-a discutat despre conversiile utilizator care permit efectuarea de conversii de date din tipuri predefinite în date de tipuri abstracte şi invers, precum şi conversia obiectelor de un tip abstract în obiecte de un alt tip abstract.

În principiu, conversiile dintr-un tip predefinit într-un tip abstract se realizează cu ajutorul constructorilor, iar conversia inversă cu ajutorul supraîncărcării operatorului *cast*. Conversia dintr-un tip abstract într-un alt tip abstract se poate realiza prim ambele metode.

Ambele metode se pot aplica atît implicit cît şi explicit.

În paragraful de faţă ne referim la conversii care se pot realiza ţinînd seama de conceptul de derivare.

Fie A o clasă pentru care B este o clasă de bază. Un obiect care este o instanţiere a clasei A se converteşte în mod implicit într-un obiect al clasei B. Aceasta, deoarece obiectele clasei A sînt obiecte specializate ale clasei B.

Conversia inversă nu se poate realiza fără a fi definită printr-un constructor sau prin supraîncărcarea operatorului *cast*.

Conceptul de moştenire impune reguli de conversie şi pentru pointeri şi referinţe la obiecte ale claselor derivate şi de bază.

Fie declaraţiile:

```
A *pa;    A a;
```

```
B *pb;      B b;
```

unde:

B - Este o clasă de bază a clasei A.

Compilatorul C++ permite atribuiri de felul celor de mai jos:

```
pa = &a;
pb = &b;
```

Atribuirile de acest fel se realizează pe baza supraîncărcării implicite a operatorului adresă (& unar), care permite ca acesta să se aplice la o dată sau obiect de orice tip.

O atribuire de forma:

```
pb = &a;
```

este acceptată de compilator la fel ca şi atribuirea obiectului *a* la obiectul *b*:

```
b = a;
```

În mod analog, atribuirea:

```
pb = pa;
```

este corectă.

În general, un pointer sau o referinţă la un obiect al unei clase derivate se poate atribui la un pointer sau o referinţă la un obiect al unei clase de bază a clasei derivate respective. Atribuirile de acest gen se realizează prin conversii implicite corespunzătoare. În schimb atribuirile:

```
pa = &b;
pa = pb;
```

sînt eronate. Ele pot fi acceptate de compilator dacă se utilizează expresii *cast* care realizează conversii explicite de pointeri:

```
pa = (A *)&b;
pa = (A *)pb;
```

Conversiile de acest fel sînt utile cînd se creează şi prelucrează obiecte de tip *colecţie*, colecţia fiind un set de obiecte de diferite tipuri abstracte derivate dintr-un tip abstract de bază.

29.3. Redefinirea datelor membru ale unei clase de bază într-o clasă derivată

O dată membru a unei clase de bază se poate redefini ca dată membru a

unei clase derivate.

Exemplu:

```
class B {
 protected:
    double x;
    double y;
 public:
    B(double xx=0,double yy=0)
    {
       x = xx;
       y = yy;
    }
    ...
};
class A:public B {
 protected:
    double x;
    double y;
 public:
    B(double dx=0,double dy=0,double bx=0,
      double by=0):B(bx,by)
    {
       x=dx;  // x este data membru a clasei derivate
       y=dy;  // y este data membru a clasei derivate
    }
    void afis() const;
    ...
};
```

Funcţia membru *afis* se apelează pentru obiecte ale clasei derivate. Ea afişează datele membru ale clasei derivate, precum şi datele membru moştenite de la clasa de bază B. Pentru a face distincţie între datele membru ale clasei derivate şi cele ale clasei de bază, se foloseşte operatorul de rezoluţie. Astfel:

B::x - Este dată membru a clasei B.

x - Este dată membru a clasei A.

Putem defini funcţia membru *afis* ca mai jos:

```
void A::afis() const
{
 printf("xbaza=%g\tybaza=%g\n",B::x,B::y);
 printf("xderivat=%g\tyderivat=%g\n",x,y);
```

```
}
```

29.4. Supraîncărcarea funcţiilor membru ale unei clase de bază într-o clasă derivată

Funcţiile membru ale claselor de bază sînt moştenite de clasele derivate. Astfel, dacă B este o clasă de bază pentru clasa A şi *f* este o funcţie membru a clasei B, atunci *f* este şi o funcţie membru a clasei A. Aceasta înseamnă că funcţia *f* poate fi utilizată atît cu obiecte ale clasei B, cît şi cu obiecte ale clasei A:

```
A a;
B b;
```

`b.f(...)` şi `a.f(...)` sînt apeluri corecte ale lui *f*.

În mod analog, funcţiile prieten ale clasei de bază sînt funcţii prieten şi pentru clasa derivată.

Moştenirea funcţiilor membru şi prieten are ca efect faptul că supraîncărcarea operatorilor pentru clasa de bază este valabilă şi pentru o clasă derivată a clasei de bază respective.

Cu toate acestea, nu totdeauna o funcţie moştenită de la o clasă corespunde pentru a fi apelată cu obiecte ale unei clase derivate. În astfel de situaţii, funcţia respectivă se va supraîncărca pentru clasa derivată. La supraîncărcarea pentru clasa derivată se poate păstra nu numai numele funcţiei, ci chiar şi numărul şi tipul parametrilor. Funcţiile supraîncărcate în acest fel se selectează nu numai după numărul şi tipul parametrilor ci şi după obiectul pentru care sînt apelate. Cu alte cuvinte, reluînd exemplul de mai sus, la apelul:

```
b.f(...)
```

se apelează funcţia membru *f* a clasei de bază B, iar la apelul:

```
a.f(...)
```

se apelează funcţia membru *f* a clasei A dacă *f* este supraîncărcată pentru clasa A şi funcţia membru *f* a clasei de bază B dacă *f* nu este supraîncărcată pentru clasa A.

O problemă care apare în acest caz este aceea cînd în corpul funcţiei membru *f*, supraîncărcată pentru clasa A, se doreşte apelul funcţiei membru *f* a clasei B:

```
A::f(...)
{
 ...
 // in acest punct este nevoie de apelul functiei membru f a clasei B
 ...
}
```

Dacă în corpul funcţiei *f* utilizăm un apel obişnuit:

```
f(...)
```

atunci acesta se realizează pentru obiectul curent şi conduce la un apel recursiv al funcţiei *f*. Pentru a apela funcţia membru *f* a clasei de bază B pentru obiectul curent care este o instanţiere a clasei derivate A, folosim operatorul de rezoluţie:

```
B::f(...)
```

Deci, corpul funcţiei *f* se va defini astfel:

```
A::f(...)
{
 ...
 B::f(...)  // se apeleaza functia membru f a clasei de
            // baza B pentru obiectul curent care este
            // instantiere a clasei A derivata din B

}
```

Exemplu:

```
class B {
 protected:
     double x;
     double y;
 public:
     B(double bx=0,double by=0)
     {
         x=bx;
         y=by;
     }

 void afis() const
     {
         printf("bx=%g\by=%g\n",x,y);
     }
     ...
};
```

```
class A:public B {
 protected:
    double xx;
    double yy;
 public:
    A(double dx=0,double dy=0,double bx=0,
      double by=0):B(bx,by)
    {
       xx=dx;
       yy=dy;
    }
    void afis() const;
    ...
};
```

Funcţia membru *afis*, a clasei de bază B, este supraîncărcată pentru clasa derivată A. Funcţia *afis* aplicată la o instanţiere a clasei B va afişa valorile lui *x* şi *y*.

Funcţia *afis* aplicată la o instanţiere a clasei A va afişa atît valorile lui *x* şi *y* ale obiectului curent cît şi valorile lui *xx* şi *yy* ale aceluiaşi obiect. De aceea, la definirea funcţiei *afis* pentru clasa A vom apela funcţia *afis* membru a clasei B.

```
void A::afis() const
{
 B::afis();   //afiseaza pe x si y
 printf("dx=%g\tdy=%g\n",xx,yy);
}
```

Exemple de apel a funcţiei *afis*:

```
B b(1,2);         //b.x=1, b.y=2
A a(1,2,3,4);     //a.x=3, a.y = 4, a.xx=1, a.yy=2
b.afis();         //se apeleaza functia membru afis a clasei B
```

Se afişează:

```
bx=1  by=2
a.afis();   //se apeleaza functia membru afis a clasei A
```

Se afişează:

```
bx=3 by=4
dx=1 dy=2
```

În legătură cu supraîncărcarea operatorului de atribuire pentru clasa

derivată amintim următoarele:

1. Dacă operatorul de atribuire nu este supraîncărcat nici pentru clasa derivată şi nici pentru clasele de bază, atunci la atribuirea a două obiecte ale clasei derivate se realizează o copiere bit cu bit a obiectului din dreapta semnului de atribuire (obiectul sursă) în obiectul din stînga semnului de atribuire (obiectul destinaţie).
2. Dacă operatorul de atribuire nu este supraîncărcat pentru clasa derivată, dar este supraîncărcat pentru una sau mai multe clase de bază, atunci pentru datele membru moştenite de la clasele de bază care au operatorul de atribuire supraîncărcat se apelează în mod automat funcţiile de supraîncărcare a operatorului de atribuire corespunzătoare, iar pentru restul datelor membru se face o copiere bit cu bit.
3. Dacă operatorul de atribuire este supraîncărcat pentru clasa derivată, atunci funcţia care supraîncarcă operatorul respectiv realizează atribuirea obiectului sursă obiectului destinaţie pentru toate datele membru. În acest caz, nu se mai apelează automat eventualele funcţii care supraîncarcă operatorul de atribuire pentru clasele de bază.

Exerciţii:

29.7 Să se modifice clasa *punct*, definită în exerciţiul 29.1, adăugînd constructorul de copiere şi supraîncărcarea operatorului de atribuire.

FIŞIERUL BXXIX7.H

```
class punct {
 protected:
    double x0;
    double y0;
    static int rcit_double(const char *text,
                           double& d);

 public:
    punct()   // constructor implicit
    {
       x0 = 0;
       y0 = 0;
    }
    punct(double abs,double ord=0);
    punct(const punct&);    // constructor de copiere
    int citpct(const char *text);
    void afispct(const char *f) const;
        friend punct operator +(punct p1,punct p2);
```

```
        friend punct operator -(punct p1,punct p2);
        punct operator *(double d);
        friend punct operator *(double d,punct p2);
        punct operator /(double d);
        double dist(punct p2) const;
        double retabs() const;
        double retord() const;
        friend int operator == (punct p1,punct p2);
        punct operator -();

        // supraincarcarea operatorului de atribuire
        punct& operator = (const punct&);
   };
```

Funcţiile membru noi (constructorul de copiere şi supraîncărcarea operatorului de atribuire) se definesc în fişierul de extensie *CPP*, de mai jos. Funcţiile membru vechi, se includ din fişierul BXXIX2.CPP.

FIŞIERUL BXXIX7

```
#ifndef __PUNCT_H
#include "BXXIX7.H"
#define __PUNCT_H
#endif
#include "BXXIX1.CPP"  // contine definitiile functiilor membru
                       // si prieten vechi

// se definesc functiile membru noi

// constructorul de copiere
punct::punct(const punct& p)
{
     x0=p.x0;
     y0=p.y0;
}

// supraincarcarea operatorului de atribuire
punct& punct::operator = (const punct& p)
{
 if(this != &p) {
     x0=p.x0;
     y0=p.y0;
 }
 return *this;
}
```

29.8 Să se modifice clasa *gpunct*, definită în exerciţiul 29.2, adăugînd constructorul de copiere şi supraîncărcînd operatorul de atribuire.

FIŞIERUL BXXIX8.H

```
#ifndef __PUNCT_H
#include "BXXIX7.CPP"
#define __PUNCT_H
#endif
#ifndef __GRAPHICS_H
#include <graphics.h>
#define __GRAPHICS_H
#endif

class gpunct:public punct {
 protected:
     int culoare;
     int vizibil;
 public:
     gpunct(double abs=0,double ord=0,int c=WHITE);
     gpunct(const punct& p,int c=WHITE);

     // constructor de copiere
     gpunct(const gpunct&);

     void afiseaza();
     void sterge();

     // supraincarca operatorul de atribuire
     gpunct& operator = (const gpunct&);
};
```

Funcţiile membru noi se definesc în fişierul de extensie *CPP*, de mai jos. Celelalte funcţii se includ din fişierul BXXIX2.CPP.

FIŞIERUL BXXIX8

```
#ifndef __GPUNCT_H
#include "BXXIX8.H"
#define __GPUNCT_H
#endif
#include "BXXIX2.CPP"

inline gpunct::gpunct(const gpunct& gp):punct(gp)
// constructor de copiere
{
```

```
 culoare = gp.culoare;
 vizibil = gp.vizibil;
}

gpunct& gpunct::operator = (const gpunct& gp)
// supraincarca operatorul de atribuire
{
 if(this != &gp){
     x0=gp.x0;
     y0=gp.y0;
     culoare=gp.culoare;
     vizibil=gp.vizibil;
 }
 return *this;
}
```

Observaţie:

Constructorul de copiere al clasei *gpunct* apelează în mod explicit constructorul de copiere al clasei *punct*. În acest scop se realizează în mod implicit o conversie a referinţei la un obiect de tip *gpunct* într-o referinţă la un obiect de tip *punct*. Într-adevăr, parametrul *gp* este o referinţă la un obiect de tip *gpunct*. În antetul constructorului de copiere se face un apel al constructorului de copiere *punct* cu parametrul *gp*. Constructorul de copiere *punct* are ca parametru o referinţă la un obiect de tip *punct*. De aceea, la apelul constructorului de copiere *punct* se realizează o conversie automată a lui *gp* din referinţă la obiecte de tip *gpunct* în referinţă la obiecte de tip *punct*.

29.9 Să se modifice clasa *gtextpunct*, definită în exerciţiul 29.5, adăugînd un constructor care are doi parametri neimpliciţi, unul pentru iniţializarea datei membru *text*, iar celălalt pentru iniţializarea datelor membru ale clasei *gpunct*, un constructor de copiere şi supraîncărcînd operatorul de atribuire.

FIŞIERUL BXXIX9.H

```
#ifndef __GTEXT_H
#include "BXXIX4.CPP"
#define __GTEXT_H
#endif
#ifndef __GPUNCT_H
#include "BXXIX8.CPP"
#define __GPUNCT_H
#endif
```

```
class gtextpunct:public gpunct,public gtext {
 protected:
     int vizibil_punct_text;
 public:
     gtextpunct();
     gtextpunct(char *sir,double x=0,
              double y=0,int c=WHITE,
          int culoare=WHITE,int f=DEFAULT_FONT,
          int dir=HORIZ_DIR,int dim_car=1,
          int cadraj_o=CENTER_TEXT,
          int cadraj_v=CENTER_TEXT);
     gtextpunct(char *sir,const gpunct& gp,
          int culoare=WHITE,int f=DEFAULT_FONT,
          int dir=HORIZ_DIR,int dim_car=1,
          int cadraj_o=CENTER_TEXT,
          int cadraj_v=CENTER_TEXT);
     gtextpunct(const gtextpunct&);
     // constructor de copiere

     void afisgtextpunct(int xt,int yt);
     void stergegtextpunct(int xt,int yt);

     gtextpunct& operator = (const gtextpunct&);
     // supraincarca operatorul de atribuire
};
```

Funcţiile membru noi se definesc în fişierul de extensie *CPP*, de mai jos. Cele vechi, se includ din fişierul BXXIX5.CPP.

FIŞIERUL BXXIX9

```
#ifndef __GTEXTPUNCT_H
#include "BXXIX9.H"
#define __GTEXTPUNCT_H
#endif
#include "BXXIX5.CPP"

gtextpunct::gtextpunct(char *sir,
      const gpunct& gp,int culoare,
       int f,int dir,int dim_car,int cadraj_o,
       int cadraj_v):gpunct(gp),gtext(sir,
       culoare,f,dir,dim_car,cadraj_o,cadraj_v)
// constructor cu doi parametri neimpliciti
{
 vizibil_punct_text = 0;
```

```
}

gtextpunct::gtextpunct(const gtextpunct& gtp):
                            gpunct(gtp)
// constructor de copiere
{
 strncpy(text,gtp.text,11);
 culoare_text = gtp.culoare_text;
 font = gtp.font;
 directie = gtp.directie;
 dimensiune_caracter = gtp.dimensiune_caracter;
 cadraj_oriz = gtp.cadraj_oriz;
 cadraj_vert = gtp.cadraj_vert;
 vizibil_text = gtp.vizibil_text;
 vizibil_punct_text = gtp.vizibil_punct_text;
}

gtextpunct& gtextpunct::operator = (const
                                    gtextpunct& gtp)
// supraincarca operatorul de atribuire
{
 if(this != &gtp){
    x0 = gtp.x0;
    y0 = gtp.y0;
    culoare = gtp.culoare;
    vizibil = gtp.vizibil;
    strncpy(text,gtp.text,11);
    culoare_text = gtp.culoare_text;
    font = gtp.font;
    directie = gtp.directie;
    dimensiune_caracter = gtp.dimensiune_caracter;
    cadraj_oriz = gtp.cadraj_oriz;
    cadraj_vert = gtp.cadraj_vert;
    vizibil_text = gtp.vizibil_text;
    vizibil_punct_text = gtp.vizibil_punct_text;
 }
 return *this;
}
```

29.10 Să se scrie un program care realizează următoarele:

- citeşte coordonatele a două puncte A şi B;
- citeşte un număr diferit de -1 pe care îl atribuie variabilei *k*;
- determină punctul C care împarte segmentul AB în raportul

k;

– afişează punctele A, B, C pe ecran împreună cu notaţiile lor.

Punctele A şi B se afişează cu culoarea definită de constanta simbolică WHITE, iar punctul C folosind constanta simbolică LIGHTGREEN.

Fie *ax* şi *ay* coordonatele punctului A şi *bx* şi *by* ale lui B. Coordonatele *cx* şi *cy* ale punctului C, care împarte segmentul AB în raportul *k*, se determină cu ajutorul formulelor:

cx = (ax+k*bx)/(k+1)
cy = (ay+k*by)/(k+1)

PROGRAMUL BXXIX10

```
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#include "BXXIX9.CPP"

main()
/* - citeste coordonatele punctelor A si B;
   - citeste pe k diferit de -1;
   - determina punctul C care imparte segmentul AB in raportul k;
   - afiseaza punctele A, B si C.
*/
{
 char t[255];

 gtextpunct A("A"),B("B");
 // citeste coordonatele lui A si B

 if(A.citpct("A") == 0)
    exit(1);
 if(B.citpct("B") == 0)
    exit(1);
 double k;
```

```
// citeste numarul k
for(;;){
    printf("raportul k=");
    if(gets(t) == 0)
        exit(1);
    if(sscanf(t,"%lf",&k) == 1 && k != -1)
        break;
    printf("nu s-a tastat un numar\
             diferit de -1\n");
}

punct p((A+k*B)/(1+k));
gpunct gp(p,LIGHTGREEN);
gtextpunct C("C",gp,LIGHTGREEN);
int gd = DETECT,gm;
initgraph(&gd,&gm,"c:\\borlandc\\bgi");
A.afisgtextpunct(A.retabs()-10,A.retord());
B.afisgtextpunct(B.retabs(),B.retord()+10);
C.afisgtextpunct(C.retabs()+10,C.retord()-10);
getch();
closegraph();
}
```

29.11 Să se implementeze tipul abstract *gsegment* pentru lucrul cu segmente de dreaptă.

Un segment de dreaptă este definit de două puncte. Clasa *gsegment* este o clasă derivată din clasa *gpunct*.

Un capăt al segmentului are coordonatele *x0,y0* moştenite de la clasa *gpunct*. Celălalt capăt al segmentului este definit de data membru *pct* a clasei *gsegment* şi care este o instanţiere a clasei *punct*.

Culoarea de trasare pe ecranul grafic a segmentului este definită de data membru *culoare* moştenită de la clasa *gpunct*.

O altă dată membru este variabila *vizibil_seg*. Dacă segmentul este afişat pe ecran, *vizibil_seg* are valoarea 1, altfel are valoarea zero.

Pe ecranul grafic se pot trasa linii de grosimi definite cu ajutorul constantelor simbolice NORM_WIDTH şi THICK_WIDTH.

În mod implicit, grosimea liniei se defineşte cu ajutorul constantei simbolice NORM_WIDTH. Stilul trasării este totdeauna cel definit de constanta simbolică SOLID_LINE.

Caracteristicile curente pentru trasarea liniilor se pot determina cu

ajutorul funcţiei *getlinesettings* de prototip (vezi paragraful 19.7.):

```
void far getlinesettings(
          struct linesettingstype far *lineinfo);
```

unde structura *linesettingstype* se defineşte în fişierul *graphics.h* astfel:

```
struct linesettingstype {
 int linestyle;
 unsigned upattern;
 int thickness;
};
```

Caracteristicile respective se pot seta folosind funcţia *setlinestyle* de prototip:

```
void far setlinestyle(int stil,unsigned sablon,
                      int grosime);
```

Parametrii *stil* şi *sablon* au valorile implicite SOLID_LINE şi respectiv zero pentru obiectele de tip *gsegment* şi ele nu pot fi modificate de utilizator. În schimb, parametrul *grosime* pentru obiectele de tip *gsegment*, are valoarea implicită NORM_WIDTH, dar poate fi modificat de utilizator la instanţierea obiectelor folosind constanta simbolică THICK_WIDTH. În acest scop, clasa *gsegment* are o dată membru corespunzătoare numită *grosime*.

La instanţierea obiectelor de tip *gsegment*, constructorul defineşte coeficienţii *m*, *n* ai ecuaţiei dreptei suport a segmentului care se instanţiază.

În general, o dreaptă care nu este paralelă cu axa *Oy*, are ecuaţia:

$y = mx + n$

Un segment este paralel cu axa *Oy* dacă abscisele capetelor segmentului sînt egale. În notaţiile de mai sus, segmentul este paralel cu axa *Oy* dacă x0 este egal cu pct.x0.

Coeficienţii *m* şi *n* se determină astfel:

```
m = (pct.y0-y0)/(pct.x0-x0);
n = y0 - m*x0
```

Dacă segmentul este paralel cu axa *Oy*, atunci dreapta are ecuaţia:

$x=x0$

şi deci coeficienţii *m* şi *n* nu se mai definesc.

Pentru a putea distinge cazul segmentelor paralele cu axa *Oy* de cele care nu sînt paralele cu axa respectivă, clasa *gsegment* are o dată membru de

nume *ypar*.

Variabila *ypar* are valoarea 1 dacă segmentul este paralel cu axa *Oy* şi zero în caz contrar. Rezultă că, dreapta suport a segmentului care se instanţiază, are ecuaţia:

y = mx+n

dacă

ypar = 0

şi

x=x0

dacă

ypar = 1.

FIŞIERUL BXXIX11.H

```
#ifndef __GPUNCT_H
#include "BXXIX8.CPP"
#define __GPUNCT_H
#endif

class gsegment:public gpunct {
 protected:
     punct pct;
     int vizibil_seg;
     int grosime;
     int ypar;
     double m,n;

     void mn();   // calculeaza coeficientii m si n

  public:
     gsegment();   // constructor implicit

     gsegment(double xpct,double ypct=0,double x=0,
            double y=0,int c=WHITE,
           int lat=NORM_WIDTH);
     // constructor utilizat la initializari si conversii

     gsegment(const punct& p0,const punct& p1,
            int c=WHITE,int lat=NORM_WIDTH);
     /* constructor utilizat la initializari cind se dau
```

punctele limita ale segmentului */

```
gsegment(const gsegment&);
// constructor de copiere

gsegment& operator=(const gsegment&);
// supraincarca operatorul de atribuire

void afiseaza();
// afiseaza segmentul in mod grafic

void sterge();
// face invizibil segmentul curent

double lungime() const;
// returneaza lungimea segmentului curent

int operator == (gsegment gs) const;
/* returneaza 1 daca segmentul curent coincide cu
   obiectul gs si zero altfel */

int operator || (gsegment gs) const;
/* returneaza 1 daca segmentul curent este paralel
   cu gs si zero altfel */

int operator && (gsegment gs) const;
/* returneaza 1 daca segmentul curent este
    perpendicular pe gs si zero altfel */

int operator & (punct p) const;
/* returneaza 1 daca punctul p apartine dreptei
   suport a segmentului curent si zero altfel */

gsegment operator ^ (gpunct p) const;
/* returneaza un segment care trece prin punctul p
   si este paralel cu segmentul curent */

gsegment operator | (gpunct p) const;
/* returneaza un segment care trece prin punctul p
   si este perpendicular pe segmentul curent */
```

```
punct operator *(gsegment gs) const;
/* - returneaza punctul de intersectie dintre dreptele suport ale
     segmentului curent si gs;
   - daca segmentele nu se intersecteaza, obiectul returnat nu
     este definit */

double retm() const;
/* - returneaza m;
   - rezultatul este nedeterminat daca segmentul este
     paralel cu axa Oy */

double retn() const;
/* - returneaza n;
   - rezultatul este nedeterminat daca segmentul este
     paralel cu axa Oy */

int operator !() const;
/* returneaza 1 daca segmentul curent este paralel cu
   axa Oy si zero altfel */

gpunct retx0y0() const;
// returneaza punctul de coordonate (x0,y0)

punct retpct() const;
// returneaza punctul pct al obiectului curent

int retculoare() const;
// returneaza culoarea obiectului curent

int retgrosime() const;
// returneaza grosimea obiectului curent

void setculoare (int c);
/* seteaza culoarea obiectului curent la valoarea lui c */

void setgrosime (int g);
/* seteaza grosimea obiectului curent la valoarea lui g */
};
```

Funcţiile membru se definesc în fişierul de mai jos, de extensie *CPP*.

Anumite funcţii efectuează teste de egalitate în care intervine coeficientul *m*. Avînd în vedere faptul că *m* este de tip *double*, aceste teste sînt afectate de erorile care intervin în mod inevitabil la calculul coeficientului

m.

De exemplu, două drepte de ecuaţii:

y=m1x+n1

şi

y=m2x+n2

sînt perpendiculare, dacă:

m1*m2 = -1

Un astfel de test este afectat, de obicei, de următoarele erori:

- eroarea de calcul a coeficienţilor *m1* şi *m2*;
- eroarea de calcul a produsului m1*m2.

De aceea, testul de mai sus este normal să fie înlocuit cu următorul:

abs(m1*m2+1) < EPS

unde:

abs(x)	- Înseamnă valoarea absolută a lui x.
EPS	- Are o valoare dependentă de precizia de calcul.

În cazul de faţă, EPS = 1.0E-13.

Un test asemănător se face în cazul în care se stabileşte paralelismul a două drepte. Astfel, dreptele amintite mai sus sînt paralele, dacă m1 = m2. Acest test este afectat de erorile de calcul ale coeficienţilor *m1* şi *m2*. De aceea, testul respectiv se înlocuieşte cu următorul:

abs(m1-m2) < EPS

Alte funcţii membru care utilizează teste de acest fel sînt:

- funcţia care testează apartenenţa unui punct la o dreaptă (operatorul & supraîncărcat);
- funcţia care determină un segment perpendicular pe unul dat.

În esenţă, un test de egalitate de forma:

(1) exp1 = exp2

unde:

exp1 şi exp2 - Sînt expresii de tip *double* se înlocuieşte cu testul:

(2) abs(exp1-exp2) < EPS

Testul (1) se consideră că are valoarea *adevărat* atunci şi numai atunci cînd testul (2) are valoarea *adevărat*.

Pentru a realiza un test de forma (1) cu ajutorul relaţiei (2), se utilizează o funcţie obişnuită numită *testegal*. Ea are prototipul:

```
int testegal(double exp1,double exp2);
```

Funcţia returnează valoarea 1 dacă relaţia (2) este adevărată şi zero altfel.

Relaţia (2) nu este concludentă dacă *exp1* şi *exp2* sînt ele însele mai mici, în valoare absolută, decît EPS. De aceea, înainte de a testa relaţia (2), funcţia testează dacă aceste expresii au valoarea absolută mai mică decît EPS. În caz afirmativ, în relaţia (2), expresia respectivă se înlocuieşte cu zero.

FIŞIERUL BXXIX11

```
#ifndef __GSEGMENT_H
#include "BXXIX11.H"
#define __GSEGMENT_H
#endif
#define EPS 1.0E-13

int testegal(double exp1,double exp2)
/* returneaza 1 daca relatia:
       abs(exp1-exp2) < EPS
  este adevarata;
  zero altfel
*/
{
 double d1 = exp1 < 0 ? -exp1:exp1;     //abs(exp1)
 double d2 = exp2 < 0 ? -exp2:exp2;     //abs(exp2)

 if(d1 < EPS)
    d1 = 0;        //abs(exp1) < EPS
 else
    d1 = exp1;     //abs(exp1) >= EPS
 if(d2 < EPS)
    d2 = 0;        //abs(exp2) < EPS
 else
    d2 = exp2;     //abs(exp2) >= EPS
 if((d1 = d1-d2) < 0)
    d1 = -d1;      //d1 = abs(exp1-exp2)
 return d1 < EPS;
```

```
}   // sfirsit testegal

void gsegment::mn()
/* - determina coeficientii m si n ai ecuatiei dreptei care trece prin
     punctele:
            (x0,y0) - punct mostenit;
            (pct.x0,pct.y0) punct membru;
   - daca segmentul curent este paralel cu axa Oy, atunci m si n
     sint nedefiniti, iar data membru ypar are valoarea 1;
   - ypar are valoarea -1 daca punctele coincid (x0=pct.x0 si y0=pct.y0);
   - ypar are valoarea zero in restul cazurilor */
{
 if(x0 != pct.retabs()) {
 // se determina coeficientii m,n si ypar=0

      m=(pct.retord()-y0)/(pct.retabs()-x0);
      n=y0-m*x0;
      ypar=0;
 }
 else
      if(y0 != pct.retord())
      // segment paralel cu Oy
           ypar = 1;
      else    // punctele coincid
           ypar = -1;
}  // sfirsit mn

gsegment::gsegment():pct(639,0)       // constructor implicit
{
 // se instantiaza un obiect implicit si anume axa Ox

 x0 = y0 = 0;
 grosime = NORM_WIDTH;
 m=n=0;
 vizibil = 0;   // data mostenita
 vizibil_seg=0;   // segmentul nu este vizibil la instantiere
 ypar=0;    // segmentul are capetele distincte si nu este
            // paralel cu Oy
}  // sfirsit constructor implicit
```

```
inline gsegment::gsegment(double xpct,double ypct,
                   double x,double y,int c,int lat):
        gpunct(x, y,c),pct(xpct,ypct),grosime(lat)
// constructor utilizat pentru initializari si conversii
{
 mn();   // determina m, n si ypar
 vizibil_seg=0;   //segmentul nu este vizibil la instantiere
}  // sfirsit constructor

inline gsegment::gsegment(const punct& p0,
                   const punct& p1,int c,int lat):
        gpunct(p0,c),pct(p1),grosime(lat)
/* constructor utilizat la instantieri cind se dau punctele limita ale
 segmentului */
{
 mn();    // determina m, n si ypar
 vizibil_seg=0;   // segmentul nu este vizibil la instantiere
}   // sfirsit constructor

gsegment::gsegment(const gsegment& gs):gpunct(gs)
// constructor de copiere
{
 pct = gs.pct;
 grosime = gs.grosime;
 ypar = gs.ypar;
 vizibil_seg=gs.vizibil_seg;
 m=gs.m;
 n=gs.n;
}   // sfirsit constructor de copiere

gsegment& gsegment::operator=(const gsegment& gs)
// supraincarca operatorul de atribuire
{
 if(this != &gs) {   /* obiectul din dreapta semnului =
                      difera de cel din stinga */
    x0=gs.x0;
    y0=gs.y0;
    culoare=gs.culoare;
    vizibil=gs.vizibil;
    double punct::*p;
    p = &punct::x0;
    pct.*p=gs.pct.*p;
    p = &punct::y0;
```

```
    pct.*p=gs.pct.*p;
    grosime=gs.grosime;
    vizibil_seg=gs.vizibil_seg;
    ypar=gs.ypar;
    m=gs.m;
    n=gs.n;
  }
 return *this;
}   //sfirsit operator =

void gsegment::afiseaza()   //afiseaza segmentul curent
{
 struct linesettingstype temp;

 //se salveaza caracteristicile curente
 getlinesettings(&temp);
 int c=getcolor();

 //se seteaza caracteristicile obiectului curent
 setlinestyle(SOLID_LINE,0,grosime);
 setcolor(culoare);

 //se afiseaza segmentul curent
 double punct:: *p;
 double punct:: *q;
 p = &punct::x0;
 q = &punct::y0;
 int x1,y1;
 getaspectratio(&x1,&y1);
 double f = (double)x1/y1;

 line(x0,y0*f,pct.*p,pct.*q*f);
 vizibil_seg=1;   //segmentul este vizibil
 vizibil=1;

 //se refac caracteristicile salvate
 setlinestyle(temp.linestyle,
              temp.upattern,temp.thickness);

 setcolor(c);
}   //sfirsit afiseaza
```

```
void gsegment::sterge()   // face invizibil segmentul curent
{
 int temp=culoare;    // salveaza culoarea segmentului curent

 // culoarea obiectului curent devine egala cu cea de fond
 culoare=getbkcolor();
 afiseaza();   /* segmentul curent se afiseaza cu culoarea de
                  fond, devenind invizibil */
 culoare =temp;  /* se reface culoarea obiectului curent */
 vizibil_seg=0;
 vizibil=0;
}   // sfirsit sterge

inline double gsegment::lungime() const
// returneaza lungimea segmentului curent
{
 return pct.dist(punct(x0,y0));
}   // sfirsit lungime

inline int gsegment::operator ==
                (gsegment gs) const
/* returneaza 1 daca segmentul curent coincide cu obiectul gs
   si zero altfel */
{
 return pct == gs.pct && x0 ==
                 gs.x0 && y0 == gs.y0;
}   // sfirsit operator ==

int gsegment::operator || (gsegment gs) const
/* returneaza 1 daca segmentul curent este paralel
   cu gs si zero altfel */
{
 if(ypar == 0 && gs.ypar == 0)
     return testegal(m,gs.m);
 if(ypar == 1 && gs.ypar == 1)
 // ambele segmente sint paralele cu Oy
     return 1;
 return 0;   // segmentele nu sint paralele
}   // sfirsit operator ||
```

```
int gsegment::operator && (gsegment gs) const
/* returneaza 1 daca segmentul curent este perpendicular pe gs
   si zero altfel */
{
 if(ypar == 0 && gs.ypar == 0)
     return testegal(m*gs.m,-1.0);
 if(ypar && gs.ypar)

 /* - segmentele sint paralele cu Oy sau cel putin unul nu este definit;
    - in ambele situatii nu sint perpendiculare. */
     return 0;

 if(ypar == 1 && gs.ypar == 0)

 /* segmentul curent este paralel cu Oy */
     return testegal(gs.m,0);

 /* daca testegal returneaza valoarea 1 inseamna ca gs.m=0,
    deci segmentul curent este perpendicular pe gs */

 if(ypar == 0 && gs.ypar == 1)

 /* segmentul gs este paralel cu Oy */
     return testegal(m,0);

 // in celelalte cazuri segmentele nu sint perpendiculare
 return 0;

}   // sfirsit operator &&

int gsegment::operator & (punct p) const
/* returneaza 1 daca punctul p apartine dreptei suport a
   segmentului curent si zero altfel */
{
 double punct::*u;
 double punct::*v;
 u = &punct::x0;
 v = &punct::y0;

 if(ypar == 0) { // segmentul curent nu este paralel cu oy
     double exp1=p.*u;
     exp1=m*exp1+n;
     double exp2=p.*v;
```

```
    return testegal(exp1,exp2);
  }

 if(ypar == 1)   //segmentul curent este paralel cu Oy

 /* punctul p apartine dreptei suport daca p.x0=x0 deoarece
    dreapta suport are ecuatia x=x0 */
    return testegal(x0,p.*u);

 // in celelalte cazuri segmentul curent nu este definit
 return 0;

}  // sfirsit operator &

gsegment gsegment::operator ^ (gpunct p) const
/* - returneaza un segment a carui dreapta suport trece prin punctul p si
     este paralela cu segmentul curent;
   - p este un capat al segmentului returnat;
   - celalalt capat al segmentului se determina astfel:
          - daca Oy este paralela cu dreapta suport a segmentului
            determinat, se considera punctul de coordonate
            (p.x0, p.y0+10);
          - altfel se considera punctul de pe dreapta determinata
            de abscisa p.x0+10;
         - culoarea segmentului este aceeasi cu a puctului p, iar
            grosimea este cea implicita: NORM_WIDTH.
*/
{
 double gpunct::*u;
 double gpunct::*v;
 u = &gpunct::x0;
 v = &gpunct::y0;
 int gpunct::*w;
 w = &gpunct::culoare;

 if(ypar == 0) {
 /* - segmentul curent nu este paralel cu Oy;
    - coeficientii m,n sint determinati pentru segmentul curent;
    - segmentul paralel cu el are dreapta suport de ecuatie:
          y = mx + nn;
     coordonatele punctului p verifica aceasta ecuatie, deci:
          nn = p.y0-m*p.x0
 */
```

```
    double nn = p.*v-m*p.*u;
    double a = p.*u + 10;
    double b = m*(p.*u + 10 ) + nn;
    double c = p.*u;
    double d = p.*v;
    int e = p.*w;

    gsegment r(a,b,c,d,e);
    return r;
 }
```

/* - segmentul curent este paralel cu Oy sau nu este definit;
  - segmentul paralel cu el si care trece prin punctul p are drept capete punctele de coordonate (p.x0,p.y0) si (p.x0,p.y0+10). */

```
 double a = p.*u;
 double b = p.*v + 10;
 double c = p.*v;
 int d = p.*w;

 gsegment r(a,b,a,c,d);
 return r;

}   // sfirsit operator ^

gsegment gsegment::operator | (gpunct p) const
```

/* - returneaza un segment a carui dreapta suport este perpendiculara pe segmentul curent si trece prin punctul p;
  - p este un capat al segmentului;
  - celalalt capat al segmentului se determina astfel:
    - daca Ox nu este paralela cu segmentul curent, atunci se considera punctul de abscisa p.x0+10 de pe dreapta suport a segmentului care se determina;
    - altfel se considera punctul de coordonate (p.x0,p.y0+10);
  - culoarea segmentului este aceeasi cu a punctului p, iar grosimea este cea implicita: NORM_WIDTH.
*/

```
{
 double gpunct::*u;
 double gpunct::*v;
 int gpunct::*w;
 u = &gpunct::x0;
 v = &gpunct::y0;
 w = &gpunct::culoare;
```

```
if(ypar == 0 && testegal(m,0)) {
//segmentul curent este paralel cu axa Ox

    double a = p.*u;
    double b = p.*v + 10;
    double c = p.*v;
    int d = p.*w;

    gsegment r(a,b,a,c,d);
    return r;
}

//segmentul curent nu este paralel cu Ox

if(ypar == 1 || ypar == -1) {
/* segmentul curent este paralel cu Oy sau este nedefinit;
   dreapta perpendiculara pe el are ecuatia:
          y = constant;
  segmentul rezultat are drept capete punctele de coordonate:
          (p.x0,p.y0) si (p.x0+10,p.y0)
*/

    double a = p.*u + 10;
    double b = p.*v;
    double c = p.*u;
    int d = p.*w;

    gsegment r(a,b,c,b,d);
    return r;
}
/* - segmentul curent nu este paralel cu nici una dintre axele Ox sau Oy;
   - coeficientii m si n sint determinati pentru segmentul curent
    si m != 0;
   - dreapta perpendiculara pe segmentul curent are coeficientul
    unghiular egal cu -1/m;  ea are ecuatia:
          y = (-1/m)*x+nn
   - coeficientul nn se determina din conditia ca dreapta sa treaca prin
    punctul (p.x0,p.y0)
*/

double nn = p.*v+(1/m)*p.*u;
double a = p.*u + 10;
```

```
 double b = (-1/m)* a + nn;
 double c = p.*u;
 double d = p.*v;
 int e = p.*w;

 gsegment r(a,b,c,d,e);
 return r;
}   //sfirsit operator |

punct gsegment::operator * (gsegment gs) const
/* - returneaza punctul de intersectie dintre dreptele suport ale
    segmentului curent si gs;
   - daca segmentele nu se intersecteaza, obiectul returnat nu este
    definit.
*/
{
 punct r;

 if(ypar == -1 || gs.ypar == -1)
 //cel putin unul dintre segmente nu este definit

    return r;

 if(ypar && gs.ypar)
 //segmentele sint paralele cu axa Oy

    return r;

 if(ypar == 1){
 /* - segmentul curent este paralel cu Oy;
    - dreapta suport are ecuatia:
          x = x0;
    - punctul de intersectie are coordonatele:
          (x0,gs.m*x0+gs.n)
 */
    punct r(x0,gs.m*x0+gs.n);
    return r;
   }

 if(gs.ypar == 1){
 /* - segmentul gs este paralel cu Oy;
    - dreapta suport are ecuatia:
          x = gs.x0;
```

```
   - punctul de intersectie are coordonatele:
          (gs.x0,m*gs.x0+n)
*/
    punct r(gs.x0,m*gs.x0+n);
    return r;
 }
/* - nici unul din segmente nu este paralel cu axa Oy;
   - segmentele sint paralele daca m = gs.m.
*/

 if(testegal(m,gs.m))
/* segmentele sint paralele avind coeficientii unghiulari egali */
    return r;

/* intersectia segmentelor se determina rezolvind sistemul de
   ecuatii liniare:
          y = m*x+n
          y = gs.m*x+gs.n
   avem:
          m*x+n = gs.m*x+gs.n
   sau
          x = (gs.n-n)/(m-gs.m)
*/

    double x = (gs.n-n)/(m-gs.m);
    punct rr(x,m*x+n);
    return rr;
}   // sfirsit operator *

inline double gsegment::retm() const
/* returneaza coeficientul m; daca segmentul este paralel cu
   Oy, rezultatul este nedefinit */
{
 return m;
}   // sfirsit retm

inline double gsegment::retn() const
/* returneaza coeficientul n; daca segmentul curent este paralel
   cu Oy, atunci rezultatul este nedefinit */
{
 return n;
}  // sfirsit retn
```

```
inline int gsegment::operator !() const
/* returneaza 1 daca segmentul este paralel cu axa Oy si zero altfel */
{
 return ypar;
}  // sfirsit operator !

inline gpunct gsegment::retx0y0() const
/* returneaza punctul de coordonate (x0,y0) */
{
 gpunct r(x0,y0,culoare);
 return r;
} // sfirsit retx0y0

inline punct gsegment::retpct() const
/* returneaza punctul pct al obiectului curent */
{
 return pct;
}  // sfirsit retpct

inline int gsegment::retculoare() const
/* returneaza culoarea obiectului curent */
{
 return culoare;
} // sfirsit retculoare

inline int gsegment::retgrosime() const
/* returneaza grosimea obiectului curent */
{
 return grosime;
}  // sfirsit retgrosime

inline void gsegment::setculoare (int c)
/* seteaza culoarea obiectului curent la valoarea lui c */
{
 culoare = c;
} // sfirsit setculoare

inline void gsegment::setgrosime (int g)
/* seteaza grosimea obiectului curent la valoarea lui g */
{
 grosime = g;
} // sfirsit grosime
```

29.12 Să se scrie o funcţie care citeşte coordonatele punctelor *x*, *y* şi *z* care reprezintă vîrfurile unui triunghi.

Funcţia returnează valoarea 1 dacă punctele respective nu sînt coliniare şi zero în caz contrar.

FUNCŢIA BXXIX12

```
#ifndef __GSEGMENT_H
#include "BXXIX11.CPP"
#define __GSEGMENT_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif

int cit3pct(punct& x,punct& y,punct& z)
/* - citeste coordonatele a 3 puncte de la intrarea standard;
   - returneaza 1 daca punctele nu sint coliniare si zero altfel. */
{
 char er[] = "s-a citit EOF\n";

 if(x.citpct("primul punct")==0){
     printf(er);
     exit(1);
 }
 if(y.citpct("al doilea punct")==0){
     printf(er);
     exit(1);
 }
 if(z.citpct("al treilea punct")==0){
     printf(er);
     exit(1);
 }
 /* se face testul de coliniaritate; punctele sint coliniare
    daca, de exemplu, x apartine segmentului yz */

     gsegment yz(y,z);   /* are culoarea si grosimea implicita
                                        (WHITE si NORM_WIDTH) */
 if(yz & x)   // x apartine segmentului yz
     return 0;
 // punctele x,y,z nu sint coliniare
 return 1;
}
```

29.13 Să se scrie un program care citeşte coordonatele punctelor *a,b,c* care sînt vîrfurile unui triunghi şi afişează pe ecranul grafic triunghiul respectiv împreună cu medianele sale.

Laturile triunghiului se afişează folosind culoarea implicită şi grosimea definită de constanta simbolică THICK_WIDTH.

Medianele se afişează folosind culoarea definită de constanta simbolică YELLOW şi grosimea implicită.

PROGRAMUL BXXIX13

```
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#include "BXXIX12.CPP"

main()
/* - citeste coordonatele punctelor a,b,c;
   - afiseaza triunghiul abc si medianele lui;
   - laturile au culoarea implicita si grosimea THICK_WIDTH;
   - medianele au culoarea YELLOW si grosimea implicita.
*/
{
 // citeste coordonatele virfurilor triunghiului

 punct a,b,c;

 if(cit3pct(a,b,c)==0){
    printf("punctele a,b,c sint coliniare\n");
    exit(1);
 }
 int gd=DETECT,gm;
 initgraph(&gd,&gm,"c:\\borlandc\\bgi");

 // se instantiaza laturile triunghiului
 gsegment ab(a,b,WHITE,THICK_WIDTH);
 gsegment ac(a,c,WHITE,THICK_WIDTH);
 gsegment bc(b,c,WHITE,THICK_WIDTH);

 // se afiseaza triunghiul pe ecranul grafic
 ab.afiseaza();
 ac.afiseaza();
 bc.afiseaza();
```

```
getch();
```

```
// determina medianele triunghiului
// mediana relativa la latura bc
punct mbc = (b+c)/2;
gsegment ma(a,mbc,YELLOW);
// mediana laturii ac
punct mac = (a+c) /2;
gsegment mb(b,mac,YELLOW);

// mediana laturii ab
punct mab = (a+b)/2;
gsegment mc(c,mab,YELLOW);

// afiseaza medianele
ma.afiseaza();
mb.afiseaza();
mc.afiseaza();
getch();
closegraph();
}
```

29.14 Să se scrie o funcţie care determină centrul şi raza cercului circumscris unui triunghi.

Amintim că centrul cercului circumscris unui triunghi este intersecţia mediatoarelor laturilor triunghiului respectiv.

Mediatoarea unui segment este perpendiculara pe mijlocul segmentului.

FUNCŢIA BXXIX14

```
#ifndef __GSEGMENT_H
#include "BXXIX11.CPP"
#define __GSEGMENT_H
#endif

void cerccirctr(punct x,punct y,punct z,
                punct& centru,double& r)
/* - determina centrul si raza cercului circumscris triunghiului xyz;
   - functia presupune ca x,y,z nu sint coliniare. */
{
 gsegment xy(x,y);    // latura xy a triunghiului
 gsegment xz(x,z);    // latura xz a triunghiului
```

```
   punct mxy = (x+y)/2;    // mijlocul segmentului xy
   gsegment mediatoarexy=xy|mxy; // mediatoarea segmentului
                                 // xy
   punct mxz=(x+z)/2;      // mijlocul segmentului xz
   gsegment mediatoarexz=xz|mxz; // mediatoarea segmentului
                                 // xz

   // centrul cercului este intersectia mediatoarelor laturilor
   // xy si xz
   punct c1;
   int x1,y1;

   c1=mediatoarexy*mediatoarexz;

   // raza este distanta de la centru la un virf al triunghiului
   r = c1.dist(z);
   getaspectratio(&x1,&y1);
   double f = (double)x1/y1;
   centru = punct(c1.retabs(),c1.retord()*f);
  }
```

29.15 Să se scrie un program care citeşte coordonatele a trei puncte care sînt vîrfurile unui triunghi şi afişează triunghiul respectiv împreună cu cercul circumscris acestuia.

Triunghiul se afişează folosind constantele simbolice YELLOW pentru culoare şi THICK_WIDTH pentru grosime. Cercul se trasează folosind caracteristicile implicite ale setării grafice ale ecranului.

PROGRAMUL BXXIX15

```
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#ifndef __CIT3PCT_H
#include "BXXIX12.CPP"    // cit3pct.cpp
#define __CIT3PCT_H
#endif
#ifndef __CERCCIRCTR_H
#include "BXXIX14.CPP"  // cerccirctr.cpp
#define __CERCCIRCTR_H
#endif
```

```
main()
/* - citeste coordonatele virfurilor unui triunghi;
   - afiseaza triunghiul folosind constantele simbolice YELLOW
     si THICK_WIDTH;
   - afiseaza cercul circumscris triunghiului.
*/
{
 punct a,b,c;

 if(cit3pct(a,b,c) == 0){
    printf("punctele a,b,c sint coliniare\n");
    exit(1);
 }
 int gd=DETECT,gm;
 initgraph(&gd,&gm,"c:\\borlandc\\bgi");

 // se instantiaza laturile triunghiului
 gsegment ab(a,b,YELLOW,THICK_WIDTH);
 gsegment ac(a,c,YELLOW,THICK_WIDTH);
 gsegment bc(b,c,YELLOW,THICK_WIDTH);

 // se afiseaza triunghiul abc
 ab.afiseaza();
 ac.afiseaza();
 bc.afiseaza();
 getch();
 punct centru;
 double raza;

 // se determina centrul si raza cercului circumscris triunghiului abc
 cerccirctr(a,b,c,centru,raza);

 // afiseaza cercul circumscris triunghiului
 circle(centru.retabs(),centru.retord(),raza);
 getch();
 closegraph();
}
```

29.16 O clasă importantă de probleme din geometria euclidiană este clasa referitoare la locuri geometrice. Adesea, este util a avea o imagine asupra unui loc geometric înainte de a găsi demonstraţiile necesare pentru a determina în mod riguros locul geometric respectiv. Astfel de imagini se obţin simplu folosind relaţiile existente între obiectele

geometrice (puncte, segmente, distante etc.) care intervin în problema de loc geometric.

Pentru a obţine imaginea locului geometric nu avem decît să afişăm deplasările obiectelor geometrice formulate ca date de intrare în problema de loc geometric, precum şi a obiectelor rezultat, folosind relaţiile existente între obiectele respective.

În exerciţiul de faţă se trasează un loc geometric întîlnit în geometria triunghiului.

Fie *a, b, c* vîrfurile unui triunghi şi se consideră cercul circumscris acestuia.

Se cere locul geometric al intersecţiei medianelor triunghiului dat, cînd punctele *b* şi *c* rămîn fixe, iar *a* se mişcă pe cercul circumscris triunghiului iniţial. În acest scop, se calculează centrul şi raza cercului circumscris triunghiului iniţial *abc*, ca în exerciţiul 29.15.

Coordonatele unui punct *p*(x,y), de pe acest cerc, satisfac relaţiile:

(1) x = xc+r*cos(alfa);
y = yc+r*sin(alfa);

unde:

(xc,yc)	- Sînt coordonatele centrului cercului.
r	- Este raza cercului.
alfa	- Este măsura, în radiani, a unghiului format de raza care trece prin punctul p cu axa Ox.

Într-adevăr, punctul *p* este situat la distanţa *r* de centrul de coordonate (xc,yc), oricare ar fi măsura *alfa*. Aceasta rezultă uşor din relaţia (1):

x-xc = r*cos(alfa)
y-yc = r*sin(alfa)

sau

(x-xc)**2 = r**2*cos(alfa)*cos(alfa)
(y-yc)**2 = r**2*sin(alfa)*sin(alfa)

Însumînd, se obţine:

(x-xc)**2+(y-yc)**2 = r**2

sau extrăgînd rădăcina pătrată din ambii membrii se obţine:

sqrt((x-xc)**2+(y-yc)**2) = r

În partea stîngă a semnului egal s-a obţinut distanţa de la punctul p(x,y)

la centrul cercului, de coordonate (xc,yc). Această distanţă este egală cu *r* oricare ar fi măsura unghiulară *alfa*. Cînd *alfa* variază de la 0 la 2*PI (PI = 3.1415...), atunci punctul *p* descrie cercul cu centrul de coordonate (xc,yc) şi de rază *r*.

De aici rezultă că, pentru a trasa locul geometric căutat, nu avem decît să afişăm intersecţia medianelor fiecărui triunghi care se formează menţinînd punctele *b* şi *c* fixe, iar coordonatele lui *a* se modifică cu ajutorul relaţiilor (1), variind pe *alfa* în intervalul amintit mai sus.

Alegem ca pas pentru alfa un grad sexagesimal.

Pentru a elimina deformările introduse de dimensiunea pixelilor, vom folosi coeficienţii de aspect (vezi 19.7.).

PROGRAMUL BXXIX16

```
#include <math.h>
#include <dos.h>
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#ifndef __CIT3PCT_H
#include "BXXIX12.CPP"
#define __CIT3PCT_H
#endif
#ifndef __CERCCIRCTR_H
#include "BXXIX14.CPP"
#define __CERCCIRCTR_H
#endif
#define PI 3.14159265358979

main()
/* - citeste coordonatele virfurilor unui triunghi;
   - afiseaza triunghiul folosind caracteristicile implicite;
   - afiseaza cercul circumscris triunghiului deplasind virful a
   al triunghiului;
   - afiseaza curba descrisa de intersectia medianelor triunghiurilor
   formate prin deplasarea virfului a pe cercul circumscris triunghiului
   initial; se utilizeaza culoarea definita de constanta simbolica
    YELLOW. */
{
 punct a,b,c;
```

```
if(cit3pct(a,b,c) == 0){
    printf("punctele a,b,c sint coliniare");
    exit(1);
}
gsegment ab(a,b);
gsegment ac(a,c);
gsegment bc(b,c);
int gd = DETECT,gm;
initgraph(&gd,&gm,"c:\\borlandc\\bgi");
int xa,ya;
getaspectratio(&xa,&ya);
double factor = (double)xa/ya;

// se afiseaza triunghiul initial
ab.afiseaza();
ac.afiseaza();
bc.afiseaza();
getch();

// mijlocul lui bc este fix
punct mbc = (b+c)/2;
punct centru;
double r;

// determina centrul si raza cercului circumscris triunghiului initial
cerccirctr(a,b,c,centru,r);

// se determina coordonatele centrului
double xc = centru.retabs();
double yc = centru.retord();
double frad = PI/180.0;
double alfa;
int g;
double rm=r*factor;

for(g=0;g < 360;g++){
// afiseaza virful a al triunghiului variabil pe cerc

    alfa = g*frad;   // conversie in radiani
    double x = xc+r*cos(alfa);
    double y = yc+r*sin(alfa);
    double y1 = yc+rm*sin(alfa);
    punct a(x,y);
```

```
        punct am(x,y1);
        gpunct ga(am);
        ga.afiseaza();

        /* determina intersectia medianelor pentru triunghiul curent */
        if((bc & a) == 0) {   // punctele nu sint coliniare
            gsegment ma(a,mbc);   // mediana relativa la bc
            punct mab = (a+b)/2;
            gsegment mc(c,mab);   // mediana relativa la ab
            gpunct im(ma*mc);     // intersectia medianelor
            gpunct imm(im.retabs(),
                       im.retord()*factor,YELLOW);
            imm.afiseaza();   // se afiseaza intersectia medianelor
        }
        delay(100);   // vizualizare ecran
    }  // sfirsit for
    getch();
    closegraph();
}
```

29.17 În exerciţiul 29.4, s-a definit tipul abstract *gtext* pentru a facilita afişarea de texte pe ecranul grafic. Clasa *gtext* are o funcţie membru care afişează textul obiectului curent. Funcţia are ca parametrii coordonatele punctului care defineşte începutul cîmpului în care se afişează textul. Aceleaşi coordonate se utilizează şi în cazul în care se apelează funcţia membru *stergegtext* pentru a face textul obiectului curent invizibil.

Mai jos, se defineşte clasa *gtext* ca o clasă derivată a clasei *punct*, definită în exerciţiul 29.7.

Datele membru *x0, y0*, moştenite de la clasa *punct*, definesc punctul de început al cîmpului în care se afişează textul obiectului curent.

FIŞIERUL BXXIX17.H

```
#ifndef __GRAPHICS_H
#include <graphics.h>
#define __GRAPHICS_H
#endif
#ifndef __PUNCT_H
#include "BXXIX7.CPP"
#define __PUNCT_H
#endif
```

```
class gtext:public punct {
 protected:
     int font;
     int directie;
     int dimensiune_caracter;
     int cadraj_oriz;
     int cadraj_vert;
     int culoare_text;
     char text[11];
     int vizibil_text;
 public:
     gtext();
     gtext(char *sir,double x=0,double y=0,
          int c=WHITE,
        int f=DEFAULT_FONT,int dir=HORIZ_DIR,
        int dim=1, int coriz=CENTER_TEXT,
        int cvert=CENTER_TEXT);
 void afisgtext(double dx=0,double dy=0);
 /* afiseaza textul definit de obiectul curent in cimpul care incepe
    in punctul de coordonate: (x0+dx,y0+dy) */

 void stergegtext(double dx=0,double dy=0);
 /* face invizibil textul care este afisat in cimpul care incepe in
    punctul de coordonate: (x0+dx,y0+dy) */

 gtext(char *sir,const punct& p,int c=WHITE,
     int f=DEFAULT_FONT,int dir=HORIZ_DIR,
     int dim=1,int coriz=CENTER_TEXT,
     int cvert=CENTER_TEXT);
 gtext(const gtext&);   // constructor de copiere
 gtext& operator = (const gtext&);
 // supraincarca operatorul de atribuire
};
```

Funcţiile membru se definesc în fişierul de mai jos, de extensie *CPP*.

FIŞIERUL BXXIX17

```
#ifndef __STRING_H
#include <string.h>
#define __STRING_H
#endif
#ifndef __GTEXT_H
#include "BXXIX17.H"
#define __GTEXT_H
```

```
#endif

gtext::gtext()   // constructor implicit
{
 text[0]='\0';
 x0=0;
 y0=0;
 culoare_text = WHITE;
 font = DEFAULT_FONT;
 directie = HORIZ_DIR;
 dimensiune_caracter = 1;
 cadraj_oriz = CENTER_TEXT;
 cadraj_vert = CENTER_TEXT;
 vizibil_text = 0;
}   // sfirsit constructor implicit

gtext::gtext(char *sir,double x,double y,
            int c,int f,int dir,int dim,
            int coriz,int cvert):
     punct(x,y)
/* constructor utilizat la initializari si conversie din sir de caractere
   in tipul gtext */
{
 strncpy(text,sir,11);
 text[10]='\0';
 culoare_text = c;
 font = f;
 directie = dir;
 dimensiune_caracter = dim;
 cadraj_oriz = coriz;
 cadraj_vert = cvert;
 vizibil_text = 0;
}  // sfirsit constructor

gtext::gtext(char *sir,const punct& p,int c,
        int f,int dir,int dim,int coriz,int cvert):
     punct(p)
// constructor utilizat la initializari
{
 strncpy(text,sir,11);
 text[10]='\0';
 culoare_text = c;
 font = f;
```

```
  directie = dir;
  dimensiune_caracter = dim;
  cadraj_oriz = coriz;
  cadraj_vert = cvert;
  vizibil_text = 0;
 }  //sfirsit constructor

gtext::gtext(const gtext& gt):punct(gt)
// constructor de copiere
{
 strncpy(text,gt.text,11);
 text[10]='\0';
 culoare_text = gt.culoare_text;
 font = gt.font;
 directie = gt.directie;
 dimensiune_caracter = gt.dimensiune_caracter;
 cadraj_oriz = gt.cadraj_oriz;
 cadraj_vert = gt.cadraj_vert;
 vizibil_text = gt.vizibil_text;
}  //sfirsit constructor de copiere

void gtext::afisgtext(double dx,double dy)
/* afiseaza textul definit de obiectul curent in cimpul care
   incepe in punctul de coordonate (x0+dx,y0+dy) */
{
 struct textsettingstype car_crt;
 int culoare_crt;

 // se salveaza caracteristicile curente
 gettextsettings(&car_crt);
 culoare_crt = getcolor();

 // se seteaza caracteristicile obiectului
 settextstyle(font,directie,dimensiune_caracter);
 settextjustify(cadraj_oriz,cadraj_vert);
 setcolor(culoare_text);

 // afiseaza textul
 outtextxy(x0+dx,y0+dy,text);

 // se refac caracteristicile
 settextstyle(car_crt.font,
              car_crt.direction,car_crt.charsize);
```

```
 settextjustify(car_crt.horiz,car_crt.vert);
 setcolor(culoare_crt);
 vizibil_text = 1;    // textul este vizibil
}   // sfirsit afistext

void gtext::stergegtext(double dx,double dy)
/* face invizibil textul obiectului curent care este afisat in
   cimpul care incepe in punctul de coordonate (x0+dx,y0+dy) */
{
 int ctemp=culoare_text;

 // salveaza culoarea textului obiectului curent
 culoare_text = getbkcolor();

 // culoarea obiectului curent devine culoarea fondului
 afisgtext(dx,dy);        // textul devine invizibil
 culoare_text=ctemp;      // se reface culoarea obiectului curent
 vizibil_text=0;          // textul este invizibil
}

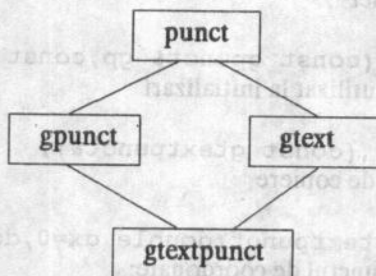
gtext& gtext::operator = (const gtext& gt)
// supraincarca operatorul =
{
 if(this != &gt){
     x0=gt.x0;
     y0=gt.y0;
     strncpy(text,gt.text,11);
     text[10]='\0';
     culoare_text = gt.culoare_text;
     font = gt.font;
     directie = gt.directie;
     dimensiune_caracter = gt.dimensiune_caracter;
     cadraj_oriz = gt.cadraj_oriz;
     cadraj_vert = gt.cadraj_vert;
     vizibil_text = gt.vizibil_text;
 }
 return *this;
}  // sfirsit operator =
```

29.18 Pentru a afişa texte împreună cu punctele, în exerciţiul 29.9 s-a introdus tipul abstract *gtextpunct*.

În exerciţiul de faţă se reimplementează tipul *gtextpunct* folosind drept

clase de bază clasa *gpunct* definită în exerciţiul 29.8 şi clasa *gtext* definită în exerciţiul 29.17.

Clasele care intervin la implementarea tipului *gtextpunct* formează o ierarhie pe care o reprezentăm în figura de mai jos.



Clasa *gtextpunct* moşteneşte datele membru *x0* şi *y0* de două ori: o dată prin intermediul clasei *gpunct* şi o dată prin intermediul clasei *gtext*. Pentru a face distincţie între ele se utilizează operatorul de rezoluţie:

gpunct::x0 şi gpunct::y0

pentru cele moştenite de la clasa *gpunct* şi

gtext::x0 şi gtext::y0

pentru cele moştenite de la clasa *gtext*.

FIŞIERUL BXXIX18.H

```
#ifndef __GPUNCT_H
#include "BXXIX8.CPP"
#define __GPUNCT_H
#endif
#ifndef __GTEXT_H
#include "BXXIX17.CPP"
#define __GTEXT_H
#endif

class gtextpunct:public gpunct,public gtext {
 protected:
    int vizibil_punct_text;
 public:
    gtextpunct();
    gtextpunct(char *sir,double xp=0,double yp=0,
             int cp=WHITE,double xt=0,double yt=0,
```

```
    int ct=WHITE,int f=DEFAULT_FONT,
    int dir=HORIZ_DIR,int dim_car = 1,
    int coriz=CENTER_TEXT,
    int cvert=CENTER_TEXT);
 /* constructor utilizat la initializari si conversie a sirurilor de caractere
   in tipul gtextpunct */

    gtextpunct(const gpunct& gp,const gtext& gt);
    // constructor utilizat la initializari

    gtextpunct (const gtextpunct&);
    // constructor de copiere

    void afisgtextpunct(double dx=0,double dy=0);
    /* - afiseaza punctul de coordonate:
              (gpunct::x0,gpunct::y0);
     - culoarea punctului este definita de data membru culoare,
       mostenita de la clasa gpunct;
     - afiseaza textul obiectului curent in cimpul care incepe in
       punctul de coordonate:
              - (gtext::x0+dx,gtext::y0+dy);
     - culoarea textului este definita de data membru culoare_text,
       mostenita de la clasa gtext;
    */

    void stergtextpunct(double dx=0,double dy=0);
    /* face invizibil punctul de coordonate:
              (gpunct::x0,gpunct::y0)
      si textul de coordonate:
              (gtext::x0+dx,gtext::y0+dy)
    */

    gtextpunct& gtextpunct::operator=(
          const gtextpunct&);
    // supraincarca operatorul =
};
```

Mai jos, se definesc funcţiile membru ale clasei *gtextpunct* într-un fişier de extensie *CPP*.

FIŞIERUL BXXIX18

```
#ifndef __GTEXTPUNCT_H
#include "BXXIX18.H"
```

```
#define __GTEXTPUNCT_H
#endif

inline gtextpunct::gtextpunct():gpunct(),gtext()
// constructor implicit
{
 vizibil_punct_text = 0;
}

inline gtextpunct::gtextpunct(char *sir,
    double xp,double yp,
    int cp,double xt,double yt,
    int ct,int f,int dir,int dim_car,
    int coriz,int cvert):
    gpunct(xp,yp,cp),gtext(sir,xt,yt,ct,
    f,dir,dim_car,coriz,cvert)
/* constructor utilizat la initializarile si conversiile sirurilor de caractere
   in obiecte de tip gtextpunct */
{
 vizibil_punct_text = 0;
}

inline gtextpunct::gtextpunct(const gpunct& gp,
    const gtext& gt):gpunct(gp),gtext(gt)
// constructor utilizat la initializari
{
 vizibil_punct_text = 0;
}

gtextpunct::gtextpunct(const gtextpunct& gtp)
// constructor de copiere
{
 strncpy(text,gtp.text,11);
 text[10] = '\0';
 double gpunct::*p;
 double gpunct:: *q;

 p = &gpunct::x0;
 q = &gpunct::y0;

 gpunct gp = gtp;
 this ->*p = gp.*p;
 this ->*q = gp.*q;
```

```
 double gtext:: *u;
 double gtext:: *v;

 u = &gtext::x0;
 v = &gtext::y0;
 gtext gt = gtp;
 this ->*u = gt.*u;
 this ->*v = gt.*v;

 culoare = gtp.culoare;
 vizibil = gtp.vizibil;
 culoare_text = gtp.culoare_text;
 dimensiune_caracter = gtp.dimensiune_caracter;
 cadraj_oriz = gtp.cadraj_oriz;
 cadraj_vert = gtp.cadraj_vert;
 vizibil_text = gtp.vizibil_text;
 vizibil_punct_text = gtp.vizibil_punct_text;
}   // sfirsit constructor de copiere

void gtextpunct::afisgtextpunct(double dx,
                                double dy)
/* - afiseaza punctul de coordonate:
        - (gpunct::x0,gpunct::y0);
   - culoarea punctului este mostenita de la clasa gpunct;
   - afiseaza textul obiectului curent in cimpul care incepe in punctul de
     coordonate:
        - (gtext::x0+dx,gtext::y0+dy);
   - celelalte caracteristici ale textului sint mostenite de la clasa gtext.
*/
{
 afiseaza();   // afiseaza punctul
 afisgtext(dx,dy);   // afiseaza textul
 vizibil_punct_text = 1;
}   // sfirsit afisgtextpunct

void gtextpunct::stergtextpunct(double dx,
                                double dy)
/* face invizibil punctul de coordonate:
        (gpunct::x0,gpunct::y0)
   si textul de coordonate:
        (gtext::x0+dx,gtext::y0+dy)
*/
```

```
{
 sterge();    // face punctul invizibil
 stergegtext(dx,dy);    // sterge textul
 vizibil_punct_text = 0;
} // sfirsit stergtextpunct

gtextpunct& gtextpunct::operator = (
            const gtextpunct& gtp)
// supraincarca operatorul =
{
 if(this != &gtp) {

     strncpy(text,gtp.text,11);
     text[10] = '\0';
     gpunct gp = gtp;
     double gpunct::*p ;
     double gpunct::*q;
     double gtext::*u;
     double gtext::*v;

     p = &gpunct::x0;
     q = &gpunct::y0;
     u = &gtext::x0;
     v = &gtext::y0;

     this ->*p = gp.*p;
     this ->*q = gp.*q;

     culoare = gtp.culoare;
     gtext gt = gtp;
     this ->*u = gt.*u;
     this ->*v = gt.*v;

     vizibil = gtp.vizibil;
     culoare_text = gtp.culoare_text;
     dimensiune_caracter = gtp.dimensiune_caracter;
     cadraj_oriz = gtp.cadraj_oriz;
     cadraj_vert = gtp.cadraj_vert;
     vizibil = gtp.vizibil;
     vizibil_text = gtp.vizibil_text;
     vizibil_punct_text = gtp.vizibil_punct_text;
 }
 return *this;
```

} // sfirsit operator =

29.19 În exercițiul 29.10 se citesc coordonatele a două puncte A și B și un număr diferit de -1 care se atribuie lui *k*. Se determină punctul C care împarte segmentul AB în raportul *k*. Apoi, se afișează punctele A, B, C, împreună cu notațiile lor.

Programul de față realizează același lucru utilizînd tipul abstract *gtextpunct* implementat în exercițiul 29.18. În acest caz, se definesc două perechi de coordonate la instanțierea obiectelor clasei *gtextpunct*: o pereche pentru coordonatele punctului și cealaltă pereche pentru a defini începutul cîmpului în care se afișează notația punctului.

La afișare, coordonatele de început ale cîmpului în care se afișează textul se pot modifica cu ajutorul parametrilor.

În programul de față nu se utilizează acești parametri (ei au valorile implicite egale cu zero).

PROGRAMUL BXXIX19

```
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#include "BXXIX18.CPP"

main()
/* - citeste coordonatele punctelor A si B;
   - citeste pe k diferit de -1;
   - determina punctul C care imparte segmentul AB in raportul k;
   - afiseaza punctele A, B si C.
*/
{
 char t[255];
 punct A,B;

 // citeste coordonatele lui A si B
 if(A.citpct("A") == 0)
     exit(1);
 if(B.citpct("B") == 0)
     exit(1);
```

```
double k;

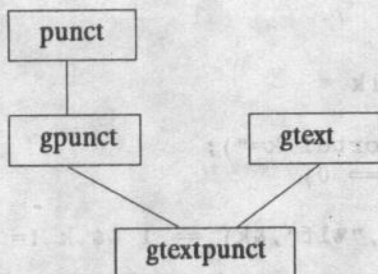
// citeste valoarea lui k
for(;;){
    printf("raportul k=");
    if(gets(t) == 0)
        exit(1);
    if(sscanf(t,"%lf",&k) == 1 && k != -1)
        break;
    printf("nu s-a tastat un numar\
            diferit de -1\n");
}
punct C = (A+k*B)/(1+k);
gpunct GA = A;   // culoarea implicita (WHITE)
gpunct GB = B;   // culoarea implicita (WHITE)
gpunct GC(C,LIGHTGREEN);
gtext GTA("A",A-10,YELLOW);
gtext GTB("B",B+punct(0,10),YELLOW);
gtext GTC("C",C+punct(10,-10),LIGHTGREEN);
gtextpunct GTPA(GA,GTA);
gtextpunct GTPB(GB,GTB);
gtextpunct GTPC(GC,GTC);
int  gd = DETECT,gm;
initgraph(&gd, &gm, "c:\\borlandc\\bgi");

GTPA.afisgtextpunct();
GTPB.afisgtextpunct();
GTPC.afisgtextpunct();
getch();
closegraph();
}
```

29.5. Clase virtuale

În exercițiul 29.9 a fost definită clasa *gtextpunct* ca o clasă derivată din clasele *gpunct* și *gtext*. Clasa *gpunct* este și ea o clasă derivată și anume, ea derivă din clasa *punct*.

Se obține o ierarhie de clase ca în figura de mai jos:



Clasa *punct* are ca date membru două date de tip double *x0* şi *y0*, care se folosesc pentru a defini poziţia unui punct pe ecranul grafic. Aceste date sînt moştenite de către clasa *gpunct* şi prin intermediul ei şi de către clasa *gtextpunct*.

Clasa *gtext* conţine date membru utilizate la afişarea textelor pe ecranul grafic.

Clasa *gtext* nu are date membru care să definească poziţia cîmpului în care să se afişeze textul obiectului de tip *gtext*. Cîmpul respectiv se defineşte la apelul funcţiei membru *afisgtext* cu ajutorul parametrilor.

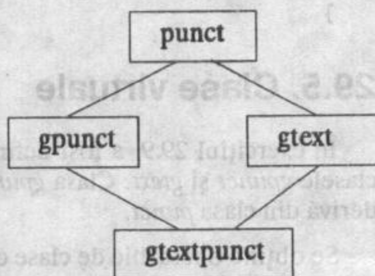
Clasa *gtextpunct* derivată ca mai sus, se utilizează pentru a afişa un punct pe ecranul grafic, împreună cu un text. Poziţia punctului este definită de datele membru *x0* şi *y0* moştenite de la clasa *punct*, iar poziţia textului se defineşte cu ajutorul parametrilor la apelul funcţiei membru *afisgtextpunct*.

Aceeaşi clasă *gtextpunct* se defineşte şi în exerciţiul 29.18, tot ca o clasă derivată din clasele *gpunct* şi *gtext*.

De data aceasta, clasa *gtext* este alta decît cea utilizată în exerciţiul 29.9.

Clasa *gtext* a fost definită în exerciţiul 29.17 ca o clasă derivată din clasa *punct*. În felul acesta, clasa *gtext* moşteneşte datele membru *x0* şi *y0* ale clasei *punct*. Aceste date definesc punctul de început al cîmpului în care se afişează textul obiectului curent. Aceste date pot fi modificate cu ajutorul parametrilor, la apelul funcţiei membru *afisgtext*.

În acest caz, se obţine următoarea ierarhie de clase:



În această ierarhie clasa *gtextpunct* moşteneşte două perechi de date membru *x0* şi *y0* de la clasa *punct*:

- una prin intermediul clasei de bază *gpunct* a clasei *gtextpunct*

şi

- una prin intermediul clasei de bază *gtext* a aceleeaşi clase *gtextpunct*.

Multiplicarea unor date membru apare în mod frecvent în cazul moştenirii multiple, ca în exemplul de mai sus. Astfel de moşteniri impun instrucţiuni suplimentare pentru a elimina ambiguităţile.

De exemplu, în cazul clasei *gtextpunct*, nu se pot face referiri simple la datele moştenite *x0*, *y0*. O pereche de date *x0* şi *y0* se foloseşte pentru a defini poziţia pentru afişarea punctului pe ecranul grafic şi acestea sînt cele moştenite cu ajutorul clasei *gpunct*, iar cealaltă pereche de date *x0* şi *y0* se foloseşte pentru a defini poziţia textului de afişat. Înlăturarea ambiguităţii, la utilizarea celor două perechi de date *x0* şi *y0* moştenite, se poate realiza aşa cum s-a procedat la definirea constructorului de copiere sau la supraîncărcarea operatorului = pentru clasa *gtextpunct* (vezi exerciţiul 29.18.).

Întrebarea care se pune în această situaţie este cea referitoare la definirea clasei *gtext*. Această clasă trebuie să fie o clasă derivată a clasei *punct* sau nu ?

În anumite situaţii este bine ca această clasă să definească poziţia în care se afişează textul obiectului curent folosind datele membru *x0*, *y0* moştenite de la clasa *punct*. Astfel de situaţii intervin atunci cînd clasa *gtext* se utilizează pentru a afişa texte independent de clasele care afişează figuri geometrice, cum sînt de exemplu clasele *gtext* şi *gsegment* definite în exemplele din acest capitol.

În caz contrar, nu este necesar ca clasa *gtext* să aibă date membru moştenite de la clasa *punct* care să precizeze poziţia textului pe ecran, poziţia respectivă fiind cea a punctului curent care se afişează împreună cu textul, eventual modificată cu ajutorul unor parametri în momentul afişării textului respectiv. De aici rezultă că ar fi bine să existe două clase pentru afişarea textelor:

- una care să moştenească datele membru *x0*, *y0* ale clasei *punct* pentru a defini poziţia textului pe ecran

şi

- una care să nu aibă nevoie de datele respective.

Limbajul C++ oferă utilizatorului o soluţie simplă pentru a realiza ambele variante cu o singură clasă *gtext*. Aceasta se realizează cu ajutorul *claselor virtuale*.

Clasele obişnuite devin virtuale în procesul de derivare. Această transformare se realizează pentru clasele de bază care intervin într-o derivare. În acest scop, numele clasei de bază, utilizat la definirea unei clase derivate, va fi precedat de cuvîntul cheie *virtual*.

Fie de exemplu, clasa A derivată din clasele de bază B1 şi B2, care la rîndul lor sînt derivate din clasa de bază B:

```
class B1:public B {
    ...
};
class B2:public B {
    ...
};

class A:public B1,public B2 {
    ...
};
```

Aşa cum s-a afirmat mai sus, clasa A moşteneşte datele membru ale lui B de două ori. Pentru a le moşteni o singură dată, transformăm clasa de bază B în clasă virtuală, ca mai jos:

```
class B1:virtual public B {
    ...
};
class B2:virtual public B {
    ...
};
```

În acest caz, clasa A moşteneşte datele membru ale lui B o singură dată. Acestea se moştenesc prin intermediul clasei de bază care este scrisă prima în lista claselor de bază. Deci, în exemplul de mai sus, datele membru ale clasei de bază virtuale B sînt moştenite prin intermediul clasei de bază B1.

Dacă clasa A se defineşte ca mai jos:

```
class A:public B2,public B1 {
    ...
};
```

atunci clasa A moşteneşte datele membru ale clasei de bază virtuale B prin intermediul clasei de bază B2.

Revenind la clasa *gtextpunct*, vom considera definiţia:

```
class gtextpunct:public gpunct,public gtext {
    ...
};
```

unde clasele *gpunct* şi *gtext* sînt derivate din clasa *punct* astfel:

```
class gpunct:virtual public punct {
    ...
};
class gtext:virtual public punct {
    ...
};
```

În felul acesta, clasa *gtextpunct* moşteneşte o singură dată datele membru *x0, y0* ale clasei *punct*. Moştenirea datelor respective se face prin intermediul clasei de bază *gpunct*.

În legătură cu clasele virtuale trebuie să amintim că există unele particularităţi în legătură cu ordinea de apel a constructorilor la instanţierea obiectelor clasei derivate.

Ca regulă generală, întîi se apelează constructorii claselor virtuale în ordinea în care au fost scrise clasele în lista claselor de bază ale clasei derivate. După apelul constructorilor claselor virtuale, se apelează constructorii claselor de bază nevirtuale în ordinea în care clasele respective apar în lista claselor de bază a clasei derivate.

Într-o ierarhie de clase, o clasă poate să apară ca şi clasă de bază de mai multe ori. Dacă o clasă de bază apare într-o ierarhie de mai multe ori, atît ca virtuală, cît şi ca nevirtuală, atunci la instanţierea unui obiect al clasei derivate se va apela constructorul clasei de bază o singură dată pentru toate apariţiile virtuale ale clasei şi de atîtea ori cîte apariţii nevirtuale are clasa de bază respectivă.

Exerciţii:

29.20 Să se modifice clasa *gtextpunct* definită în exerciţiul 29.18 folosind clasa de bază virtuală *punct*. Datele membru *x0, y0* ale clasei *punct* sînt moştenite de clasele *gpunct* şi *gtext* derivate din clasa virtuală *punct*.

Clasa *gtextpunct* este derivată, în mod obişnuit, din clasele de bază *gpunct* şi *gtext*. Ea moşteneşte datele membru *x0* şi *y0* prin intermediul clasei *gpunct*.

Funcţia membru *afisgtextpunct* a clasei *gtextpunct*, afişează pe ecranul grafic punctul de coordonate (x0,y0) moştenit prin intermediul clasei *gpunct*, precum şi textul obiectului curent. Textul se afişează într-un cîmp care începe în punctul de coordonate (x0+dx,y0+dy), unde *dx* şi *dy* sînt

parametrii funcţiei *afisgtextpunct*.

Punctul şi textul, afişate cu ajutorul funcţiei *afisgtextpunct* pot fi făcute invizibile apelînd funcţia membru *stergegtextpunct*.

FIŞIERUL BXXIX20.H

```
#ifndef __PUNCT_H
#include "BXXIX7.CPP"
#define __PUNCT_H
#endif
#ifndef __GRAPHICS_H
#include <graphics.h>
#define __GRAPHICS_H
#endif

class gpunct:virtual public punct {
 protected:
     int culoare;
     int vizibil;
 public:
     gpunct(double abs=0,double ord=0,int c=WHITE);
     gpunct(const punct&,int c=WHITE);

     // constructor de copiere
     gpunct(const gpunct&);

     void afiseaza();
     void sterge();

     // supraincarca operatorul de atribuire
     gpunct& operator = (const gpunct&);
};

#ifndef __GRAPHICS_H
#include <graphics.h>
#define __GRAPHICS_H
#endif
#define __GPUNCT_H
#include "BXXIX2.CPP"

class gtext:virtual public punct {
 protected:
     int font;
     int directie;
```

```
    int dimensiune_caracter;
    int cadraj_oriz;
    int cadraj_vert;
    int culoare_text;
    char text[11];
    int vizibil_text;
  public:
    gtext();
    gtext(char *sir,double x=0,double y=0,
          int c=WHITE,
        int f=DEFAULT_FONT,int dir=HORIZ_DIR,
        int dim=1, int coriz=CENTER_TEXT,
        int cvert=CENTER_TEXT);
    void afisgtext(double dx=0,double dy=0);
    /* afiseaza textul definit de obiectul curent in cimpul care incepe
       in punctul de coordonate
            (x0+dx,y0+dy) */

    void stergegtext(double dx=0,double dy=0);
    /* face invizibil textul care este afisat in cimpul care incepe
       in punctul de coordonate
            (x0+dx,y0+dy) */

    gtext(const gtext&);   // constructor de copiere
    gtext& operator = (const gtext&);
    // supraincarca operatorul de atribuire
};

class gtextpunct:public gpunct,public gtext {
 protected:
    int vizibil_punct_text;
 public:
    gtextpunct();   // constructor implicit
    gtextpunct(char *sir,double x=0,double y=0,
        int cp=WHITE,int ct=WHITE,
        int f=DEFAULT_FONT,
        int dir=HORIZ_DIR,int dim_car=1,
        int coriz=CENTER_TEXT,
        int cvert=CENTER_TEXT);
    /* constructor utilizat la initializari si conversii */

    gtextpunct(const gtextpunct&);
    // constructor de copiere
```

```
void afisgtextpunct(double dx,double dy);
/* - afiseaza punctul de coordonate (x0,y0);
 - afiseaza textul obiectului curent in cimpul care incepe in
      punctul de coordonate
            (x0+dx,y0+dy)
*/

void stergegtextpunct(double dx,double dy);
/* face invizibil punctul de coordonate (x0,y0) si textul din
   cimpul care incepe in punctul de coordonate (x0+dx,y0+dy)
*/

gtextpunct& operator=(const gtextpunct&);
// supraincarca operatorul =
};
```

Mai jos, se definesc funcţiile membru ale clasei *gtextpunct*, într-un fişier de extensie *CPP*.

FIŞIERUL BXXIX20

```
#include "BXXIX20.H"

inline gpunct::gpunct(const gpunct& gp):punct(gp)
// constructor de copiere
{
 culoare = gp.culoare;
 vizibil = gp.vizibil;
}

gpunct& gpunct::operator = (const gpunct& gp)
// supraincarca operatorul de atribuire
{
 if(this != &gp){
    x0=gp.x0;
    y0=gp.y0;
    culoare=gp.culoare;
    vizibil=gp.vizibil;
 }
 return *this;
}

#ifndef __STRING_H
#include <string.h>
#define __STRING_H
```

```
#endif
#define __GTEXT_H

gtext::gtext() //constructor implicit
{
 text[0]='\0';
 x0=0;
 y0=0;
 culoare_text = WHITE;
 font = DEFAULT_FONT;
 directie = HORIZ_DIR;
 dimensiune_caracter = 1;
 cadraj_oriz = CENTER_TEXT;
 cadraj_vert = CENTER_TEXT;
 vizibil_text = 0;
} //sfirsit constructor implicit

gtext::gtext(char *sir,double x,double y,int c,
        int f,int dir,int dim,int coriz,int cvert):
      punct(x,y)
/* constructor utilizat la initializari si conversie din sir de caractere
  in tipul gtext */
{
 strncpy(text,sir,11);
 text[10]='\0';
 culoare_text = c;
 font = f;
 directie = dir;
 dimensiune_caracter = dim;
 cadraj_oriz = coriz;
 cadraj_vert = cvert;
 vizibil_text = 0;
} //sfirsit constructor

gtext::gtext(const gtext& gt):punct(gt)
//constructor de copiere
{
 strncpy(text,gt.text,11);
 text[10]='\0';
 culoare_text = gt.culoare_text;
 font = gt.font;
 directie = gt.directie;
 dimensiune_caracter = gt.dimensiune_caracter;
```

```
 cadraj_oriz = gt.cadraj_oriz;
 cadraj_vert = gt.cadraj_vert;
 vizibil_text = gt.vizibil_text;
} //sfirsit constructor de copiere

void gtext::afisgtext(double dx,double dy)
/* afiseaza textul definit de obiectul curent in cimpul care incepe in
   punctul de coordonate (x0+dy,y0+dy) */
{
 struct textsettingstype car_crt;
 int culoare_crt;

 // se salveaza caracteristicile curente
 gettextsettings(&car_crt);
 culoare_crt = getcolor();

 // se seteaza caracteristicile obiectului
 settextstyle(font,directie,dimensiune_caracter);
 settextjustify(cadraj_oriz,cadraj_vert);
 setcolor(culoare_text);

 // afiseaza textul
 outtextxy(x0+dx,y0+dy,text);

 // se refac caracteristicile
 settextstyle(car_crt.font,car_crt.direction,
              car_crt.charsize);
 settextjustify(car_crt.horiz,car_crt.vert);
 setcolor(culoare_crt);
 vizibil_text = 1;    // textul este vizibil
} // sfirsit afisgtext

void gtext::stergegtext(double dx,double dy)
/* face invizibil textul obiectului curent care este afisat in cimpul care
   incepe in punctul de coordonate (x0+dx,y0+dy) */
{
 int ctemp=culoare_text;
 // salveaza culoarea textului obiectului curent

 culoare_text = getbkcolor();
 // culoarea obiectului curent devine culoarea fondului

 afisgtext(dx,dy);     // textul devine invizibil
```

```
 culoare_text=ctemp;  // se reface culoarea obiectului curent
 vizibil_text=0;      // textul este invizibil
}

gtext& gtext::operator = (const gtext& gt)
// supraincarca operatorul =
{
 if(this != &gt){
     x0=gt.x0;
     y0=gt.y0;
     strncpy(text,gt.text,11);
     text[10]='\0';
     culoare_text = gt.culoare_text;
     font = gt.font;
     directie = gt.directie;
     dimensiune_caracter = gt.dimensiune_caracter;
     cadraj_oriz = gt.cadraj_oriz;
     cadraj_vert = gt.cadraj_vert;
     vizibil_text = gt.vizibil_text;
 }
 return *this;
} // sfirsit operator =

inline gtextpunct::gtextpunct()
// constructor implicit
{
 vizibil_punct_text = 0;
} // sfirsit constructor implicit

gtextpunct::gtextpunct(char *sir,double x,
      double y,int cp,
       int ct,int f,int dir,int dim_car,
       int coriz,int cvert)
/* constructor pentru initializari si conversii */
{
 strncpy(text,sir,11);
 text[10] = '\n';
 x0 = x;
 y0 = y;
 culoare = cp;
 culoare_text = ct;
 font = f;
 directie = dir;
```

```
 dimensiune_caracter = dim_car;
 cadraj_oriz = coriz;
 cadraj_vert=cvert;
 vizibil = 0;
 vizibil_text = 0;
 vizibil_punct_text = 0;
} //sfirsit constructor pentru initializari si conversii

gtextpunct::gtextpunct(const gtextpunct& gtp)
// constructor de copiere
{
 double punct::*p;
 p = &punct::x0;
 punct pp = gtp;
 x0 = pp.*p;
 p = &punct::y0;
 y0 = pp.*p;
 culoare = gtp.culoare;
 vizibil = gtp.vizibil;
 strncpy(text,gtp.text,11);
 text[10] = '\0';
 culoare_text = gtp.culoare_text;
 font = gtp.font;
 directie = gtp.directie;
 dimensiune_caracter = gtp.dimensiune_caracter;
 cadraj_oriz = gtp.cadraj_oriz;
 cadraj_vert = gtp.cadraj_vert;
 vizibil_text = gtp.vizibil_text;
 vizibil_punct_text = gtp.vizibil_punct_text;
} //sfirsit constructor de copiere

void gtextpunct::afisgtextpunct(double dx,
                                double dy)
/* afiseaza punctul de coordonate (x0,y0) si textul obiectului curent in
   cimpul care incepe in punctul de coordonate
        (x0+dx,y0+dy) */
{
 afiseaza();  //afiseaza punctul (x0,y0)
 afisgtext(dx,dy);

 /* afiseaza textul in cimpul care incepe in punctul de coordonate
        (x0+dx,y0+dy) */
```

```
 vizibil_punct_text = 1;
} //sfirsit afisgtextpunct

void gtextpunct::stergegtextpunct(double dx,
                                  double dy)
/* face invizibil punctul si textul obiectului curent */
{
 sterge(); // face invizibil punctul
 stergegtext(dx,dy); // face textul invizibil
 vizibil_punct_text = 0;
} //sfirsit stergegtextpunct

gtextpunct& gtextpunct::operator = (
              const gtextpunct& gtp)
// supraincarca operatorul =
{
 if(this != &gtp){
    double punct::*p;
    p = &punct::x0;
    punct pp=gtp;
    x0 = pp.*p;
    p = &punct::y0;
    y0 = pp.*p;
    culoare = gtp.culoare;
    vizibil = gtp.vizibil;
    strncpy(text,gtp.text,11);
    text[10] = '\0';
    culoare_text = gtp.culoare_text;
    font = gtp.font;
    directie = gtp.directie;
    dimensiune_caracter = gtp.dimensiune_caracter;
    cadraj_oriz = gtp.cadraj_oriz;
    cadraj_vert = gtp.cadraj_vert;
    vizibil_text = gtp.vizibil_text;
    vizibil_punct_text = gtp.vizibil_punct_text;
 }
return *this;
} // sfirsit operator =
```

29.21 Să se scrie un program care realizează următoarele:

- citeşte, de la intrarea standard, coordonatele a două puncte A şi B;
- afişează punctele A şi B împreună cu numele lor;

- afişează punctele care împart segmentul AB în raportul *k*, unde *k* ia valori în intervalul [0,5;1,5] cu pasul 0,1;

- ultimul punct este afişat împreună cu litera C.

La afişarea punctelor A şi B se foloseşte culoarea implicită. Punctele care împart segmentul AB în raportul *k* se afişează folosind culoarea definită de constanta simbolică LIGHTGREEN. Litera C se afişează folosind culoarea definită de constanta simbolică YELLOW. Poziţiile literelor A, B, C se stabilesc în dependenţă de poziţiile punctelor pe care le notează, astfel:

- punctul A: x0-10,y0;

- punctul B: x0,y0+10;

- punctul C: x0,y0-10

unde:

(x0,y0) - Sînt coordonatele punctului curent.

PROGRAMUL BXXIX21

```
#ifndef __CONIO_H
#include <conio.h>
#define __CONIO_H
#endif
#ifndef __STRING_H
#include <string.h>
#define __STRING_H
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#define __STDLIB_H
#endif
#ifndef __DOS_H
#include <dos.h>
#define __DOS_H
#endif
#include "BXXIX20.CPP"

main()
/* - citeste coordonatele punctelor A si B;
   - afiseaza punctele A si B impreuna cu numele lor;
   - afiseaza punctele care impart segmentul AB in raportul k; k ia valori
     in intervalul [0,5;1,5] cu pasul 0,1; ultimul punct afisat este notat
     cu litera C */
```

```
{
 punct A,B;

 if(A.citpct("A")==0)
    exit(1);  //s-a tastat EOF
 if(B.citpct("B")==0)
    exit(1);  //s-a tastat EOF
 int gd=DETECT,gm;
 initgraph (&gd,&gm,"C:\\borlandc\\bgi");
 gtextpunct GTA("A",A.retabs(),A.retord());
 gtextpunct GTB("B",B.retabs(),B.retord());
 GTA.afisgtextpunct(-10,0);  //afiseaza A
 GTB.afisgtextpunct(0,10);   //afiseaza B
 for(double k=0.5;k < 1.6;k += 0.1){
    punct C=(A+k*B)/(1+k);
    gpunct GC(C,LIGHTGREEN);
    GC.afiseaza();
    delay(100);
    if(k < 1.5)
       continue;
    gtextpunct GTC("C",C.retabs(),
          C.retord(),LIGHTGREEN,YELLOW);
    GTC.afisgtextpunct(0,-10); //afiseaza C
    delay(100);
 }
 getch();
 closegraph();
}
```

30. FUNCŢII VIRTUALE

În capitolele 22 şi 27 s-a arătat că, o funcţie membru nestatică a unei clase se apelează, totdeauna, pentru un obiect al clasei pentru care funcţia respectivă este funcţie membru.

Legătura dintre funcţie şi obiectul pentru care se apelează funcţia se realizează cu operatorul punct (.) sau săgeată (->). Obiectul pentru care se realizează apelul se numeşte obiect curent.

Exemplu:

Fie definiţia de clasă:

```
class cl {
 public:
     ... f(...);
     ... g(...);
     ...
};
```

unde *f* şi *g* sînt funcţii membru nestatice, şi declaraţiile:

```
cl obcl;
cl *pcl;
```

Atunci funcţiile *f* şi *g* pot fi apelate numai în construcţii de felul celor de mai jos:

```
obcl.f(...)
obcl.g(...)
```

sau

```
pcl = &obcl;
pcl -> f(...)
pcl -> g(...)
```

O funcţie membru nestatică poate fi apelată în corpul unei alte funcţii nestatice fără a mai indica obiectul pentru care se face apelul. Acesta va fi în mod implicit obiectul curent. De exemplu, dacă funcţia f, din exemplul de mai sus, apelează funcţia *g* în felul următor:

```
... f(...)
{
 ...
 g(...)
 ...
```

```
}
```

atunci la apelul:

```
obcl.f(...)
```

funcţia *f* apelează funcţia *g* pentru obiectul curent, adică pentru obcl.

Acest mecanism se realizează foarte simplu cu ajutorul pointerului implicit *this* care este definit în mod implicit în corpul oricărei funcţii membru nestatice. El are ca valoare chiar adresa obiectului curent.

Înseamnă că la apelul de mai sus, în corpul funcţiei *f*, *this* are ca valoare adresa lui *obcl*. Apelul lui *g* în corpul funcţiei *f* se realizează în mod implicit sub forma:

```
this -> g(...)
```

de unde rezultă că *g* se apelează pentru obiectul curent.

Regula de apel a funcţiilor membru nestatice are şi alte posibilităţi în cazul în care clasele formează ierarhii.

Să presupunem că A este o clasă derivată din clasa de bază B:

```
class B {
 protected:
     ...
 public:
     ...
     ... fb(...);    // functie membru nestatica
     ...
};

class A:public B {
     ...
 public:
     ...
     ... fa(...);    // functie membru nestatica
     ...
};
```

Fie declaraţiile:

```
B obb;
B *pb;
A oba;
A *pa;
```

Conform celor spuse mai sus, următoarele apeluri sînt corecte:

```
obb.fb(...)
...
pb = &obb;
pb -> fb(...)
...
oba.fa(...)
...
pa = &oba;
oba -> fa(...)
...
```

Întrucît un obiect al clasei A este în acelaşi timp şi un obiect particularizat al clasei B, sînt corecte şi apelurile funcţiei *fb* cu obiecte ale clasei A:

```
oba.fb(...)
...
pa = &oba;
pa -> fb(...)
```

Corectitudinea acestor apeluri rezultă din faptul că funcţia *fb* este moştenită de clasa derivată A.

Apeluri de acest fel întîlnim în exerciţiile din capitolul 29. De exemplu, funcţia *afisgtextpunct* care este o funcţie membru nestatică a clasei *gtextpunct* se defineşte în exerciţiul 29.20, astfel:

```
void gtextpunct::afisgtextpunct(double dx,
                                double dy)
{
 afiseaza();
 afisgtext(dx,dy);
 ...
}
```

Funcţia *afiseaza* apelată în corpul funcţiei *afisgtextpunct*, este o funcţie membru nestatică a clasei *gpunct* care este o clasă de bază a clasei *gtextpunct*.

În mod analog, funcţia *afisgtext* apelată în corpul aceleeaşi funcţii, este o funcţie membru nestatică a clasei *gtext* care şi ea este o clasă de bază a clasei *gtextpunct*.

Ambele funcţii (*afiseaza* şi *afisgtext*), se apelează pentru obiectul curent care este un obiect al clasei *gtextpunct* deoarece apelurile din corpul funcţiei *afisgtextpunct* se realizează sub forma:

```
this -> afiseaza()
```

şi

```
this -> afisgtext(dx,dy)
```

Apelurile de forma:

```
obb.fa(...)
```

şi

```
pb = &obb;
pb -> fa(...)
```

sînt eronate. Într-adevăr, funcţia *fa* nu este funcţie membru a clasei B şi *fa* nu poate fi apelată pentru un obiect care este o instanţiere a clasei B.

Să observăm că, o secvenţă de forma:

```
pa = &obb;
pa -> fa(...)
```

conduce la o eroare, deoarece un pointer spre un obiect al unei clase de bază nu se poate atribui unui pointer spre o clasă derivată. Cu toate acestea, atribuirea de mai sus poate fi forţată printr-o conversie explicită a pointerilor:

```
pa = (A *)&obb;
```

În continuare se poate realiza apelul:

```
pa -> fa(...)
```

Apelurile de acest fel nu sînt recomandate deoarece, de obicei, conduc la erori.

O altă problemă care prezintă interes într-o ierarhie de clase este cea cu privire la supraîncărcarea funcţiilor.

Am văzut că în limbajul C++ se pot defini mai multe funcţii cu acelaşi nume, diferenţa dintre ele făcîndu-se după numărul şi/sau tipul parametrilor.

Se pot defini funcţii membru nestatice cu acelaşi nume şi chiar şi cu aceeaşi listă de parametri (vidă sau nu) pentru clase diferite. Aceasta nu constituie o ambiguitate deoarece funcţiile nu se apelează independent, ci legate de un obiect şi în felul acesta se selectează, în mod unic, funcţia membru care se apelează. Numai funcţiile membru ale aceleeaşi clase sau cele care nu sînt funcţii membru trebuie să difere prin numărul şi/sau tipurile parametrilor.

Exemplu:

Pentru clasele *complex* şi *rational* se defineşte funcţia *afiseaza*, fără parametri:

```
class complex {
    double real;
    double imag;
  public:
    complex(double x=0,double y=0)
    {
        real=x;
        imag=y;
    }

    void afiseaza()
    {
        printf("%g+i*(%g)\n",real,imag);
    }
    ...
};
class rational {
    long numarator;
    long numitor;
  public:
    rational(long a=0,long b=1)
    {
        numarator = a;
        if(b)
            numitor = b;
        else
            numitor = 1;
    }
    void afiseaza()
    {
        printf("(%ld)/(%ld)\n",numarator,numitor);
    }
    ...
};
```

Fie instanţierile:

```
complex z(1,2);   //z= 1+2i
```

```
rational f(1,2); //f=1/2
...
a.afiseaza();       //se afiseaza 1+i*(2)
f.afiseaza();       //se afiseaza (1)/(2)
```

Clasele *complex* şi *rational* au, fiecare, cîte o funcţie membru *afiseaza* nestatică. Ele nu au parametri. Se apelează în legătură cu un obiect şi acesta permite selectarea lor. Astfel, dacă funcţia *afiseaza* se apelează pentru un obiect *complex*, atunci se apelează funcţia membru *afiseaza* a clasei *complex*. În mod analog, la un apel al funcţiei *afiseaza* cu un obiect al clasei *rational*, se apelează funcţia membru *afiseaza* a clasei *rational*.

La definirea clasei *complex* se pot defini şi alte funcţii *afiseaza*, dar acestea vor avea parametri. De exemplu, putem defini pentru clasa *complex*, alături de funcţia *afiseaza* de mai sus, următoarea funcţie membru:

```
void afiseaza(char *f)
{
 if(f && *f)
    printf(f,real,imag);
 else
    printf("%g+i*(%g)\n",real,imag);
}
```

În cazul ierarhiilor de clase se moştenesc în clasa derivată funcţii membru din clasa de bază. Aceasta ne permite să moştenim supraîncărcările operatorilor definite în clasele de bază. De asemenea, se pot supraîncărca, în clasa derivată, funcţii definite în clasele de bază.

În acest caz, funcţiile membru ale clasei derivate care supraîncarcă funcţii din clasele de bază pot diferi de acestea prin număr şi/sau tip de parametri, dar pot şi să aibă aceeaşi listă de parametri (de exemplu, este frecvent cazul cînd lista parametrilor este vidă).

În cazul cînd funcţiile au aceeaşi listă de parametri,apar situaţiile care se precizează mai jos.

Fie clasele B şi A definite astfel:

```
class B {
    ...
public:
    ... f(...);
    ...
};
```

```
class A:public B {
    ...
 public:
    ... f(...);
    ...
};
```

Funcţia *f* este definită în clasa de bază B şi este supraîncărcată cu aceeaşi listă de parametri (care poate fi şi vidă) în clasa derivată A.

Fie declaraţiile:

```
B obb;
B *pb;
A oba;
A *pa;
...
```

La apelurile:

```
obb.f(...)
```

şi

```
pb -> f(...)
```

se apelează funcţia membru *f* a clasei B.

La apelurile:

```
oba.f(...)
```

şi

```
pa -> f(...)
```

se apelează funcţia membru *f* a clasei A.

Selectările de mai sus ale funcţiei *f* se fac în mod obişnuit ţinînd seama de obiectul pentru care se face apelul şi anume: se selectează funcţia membru a clasei căreia îi aparţine obiectul curent.

Se pune acum problema apelării funcţiei membru *f* a clasei de bază B, pentru un obiect al clasei derivate A.

Am văzut că acest lucru, în general, este posibil (vezi apelurile funcţiei *fb* de mai sus) şi anume atunci cînd funcţiile membru ale celor două clase au nume diferite. În cazul de faţă se utilizează operatorul de rezoluţie:

```
oba.B::f(...)
```

La acest apel se va utiliza, pentru obiectul *oba* al clasei derivate A,

funcţia membru *f* a clasei de bază B.

Exemplu:

```
class punct {
 protected:
     double x0,y0;
 public:
     punct(double a,double b)
     {
         x0 = a;
         y0 = b;
     }
     ...
     void afiseaza()
     {
         printf("(%g,%g)\n",x0,y0);
     }
     ...
};

class segment:public punct {
 protected:
     punct pct;
 public:
     segment(double a,double b,double c,double d) :
         punct(a,b),pct(c,d)
     {
     }
     ...
     void afiseaza()
     {
         punct::afiseaza();
         pct.afiseaza();
     }
     ...
};
```

Funcţia *afiseaza* este definită ca funcţie membru în clasa de bază *punct* şi este supraîncărcată în clasa derivată *segment*.

Fie declaraţiile:

```
punct a(100,100);
segment ab(20,20,200,200);
```

Apelul:

```
a.afiseaza()
```

apelează funcţia membru *afiseaza* a clasei *punct*.

Ca rezultat se afişează: (100,100)

Apelul:

```
ab.afiseaza()
```

apelează funcţia membru *afiseaza* a clasei *segment*. Ca rezultat se afişează:

(20,20)
(200,200)

Apelul:

```
ab.punct::afiseaza()
```

apeleaza funcţia membru *afiseaza* a clasei *punct*. Ca rezultat se afişează datele membru moştenite, adică:

(20,20)

Să observăm modul de realizare al funcţiei membru *afiseaza* a clasei *segment*.

Prima instrucţiune este:

```
punct::afiseaza();
```

Ea trebuie interpretată sub forma:

```
this -> punct::afiseaza()
```

adică se apelează,pentru obiectul curent (care este de tip *segment*), funcţia membru *afiseaza* a clasei *punct*.

Ca rezultat al acestui apel se va afişa:

(x0,y0)

adică datele membru moştenite de la clasa *punct*.

Cea de a doua instrucţiune:

```
pct.afiseaza();
```

se interpretează sub forma:

```
this -> pct.afiseaza();
```

Aceasta înseamnă că funcţia *afiseaza* se apelează pentru data membru *pct* a obiectului curent. Acesta este de tip *punct* şi de aceea se apelează

funcţia membru *afiseaza* a clasei *punct*.

Să observăm că funcţia membru *afiseaza* a clasei *segment*, definită ca mai jos:

```
void afiseaza()
{
 afiseaza();
 pct.afiseaza();
}
```

reprezintă un apel recursiv infinit al funcţiei membru *afiseaza* a clasei *segment*.

Într-adevăr, prima instrucţiune se realizează sub forma:

```
this -> afiseaza();
```

adică se apelează funcţia *afiseaza* pentru obiectul curent. Acesta este o instanţiere a clasei *segment*, deci apelul respectiv este o reapelare recursivă a funcţiei *afiseaza* a clasei *segment*.

Ca o concluzie generală a celor spuse mai sus este faptul că selectările funcţiilor supraîncărcate se tratează, în toate cazurile care au fost analizate, în momentul *compilări*. Ele nu se modifică la execuţie şi de aceea, astfel de selectări se numesc *statice*. O altă interpretare este aceea că funcţia apelată este legată de obiectul pentru care se apelează *timpuriu* (în faza compilării, spre deosebire de legătura *întîrziată* care se face la execuţie).

Există cazuri cînd selectările funcţiilor membru nu se pot face la compilare. Astfel de situaţii apar frecvent la instanţierea şi utilizarea obiectelor de tip *container*.

Un obiect de tip *container* este un set de obiecte neomogene (care nu sînt de un acelaşi tip) care aparţin claselor unei ierarhii. Obiectele de acest fel se implementează prin tablouri sau liste care, de obicei, se numesc *eterogene*.

Un tablou eterogen este un tablou obişnuit de pointeri spre tipul situat în rădăcina ierarhiei.

Fie B clasa situată în vîrful ierarhiei de clase şi A1,A2,...,Am clase situate pe diferite nivele ale ierarhiei. Atunci, un tablou eterogen utilizat pentru a implementa un container de obiecte care aparţin claselor ierarhiei, se declară astfel:

```
B *tab[max];
```

unde:

max - Este o constantă care defineşte numărul maxim de

obiecte ale containerului.

Definim clasa *container*:

```
class container {
    B *tab[100];
    int n;    // indicele elementului liber
  public:
    B()
    {
        n = 0;
    }

    int atrib(B *bp)
    {
        if(n < 100){
            tab[n++] = bp;
            return 1;
        }
        else
        return 0;  // container plin
    }

    B *acceselem(int i)
    {
        if(0 <= i && i < n)
            return tab[i];
        else
            return 0;    // pointerul nul
    }
};
```

Elementele lui *tab* sînt pointeri spre obiecte de tip B. Cum A1,A2,...,Am sînt clase derivate direct sau indirect din B, rezultă că obiectele claselor A1,A2,...,Am sînt obiecte particulare ale clasei B. De aceea, un pointer spre Ai se atribuie în mod obişnuit unui pointer spre B. Aceasta înseamnă că funcţia *atrib* poate fi apelată cu parametri efectivi care sînt pointeri spre oricare din obiectele claselor ierarhiei. În particular, funcţia *atrib* poate avea ca parametri efectivi expresii de forma:

&OBAi, i = 1,2,...,m

unde prin OBAi am notat un obiect al clasei Ai.

Fie instanţierea:

```
container AB;
```

Un apel de forma:

```
AB.atrib(&OBAi);
```

atribuie elementului:

```
AB.tab[n]
```

pointerul spre obiectul OBAi al clasei Ai. Totodată se incrementează valorea lui *n*. Cu alte cuvinte, apelul funcţiei *atrib* de mai sus, poate fi considerat că pune obiectul OBAi,al clasei Ai, în containerul AB.

În containerul AB se pot pune obiecte ale oricăreia din clasele ierarhiei.

Funcţia *acceselem* returnează pointerul spre elementul tab[i]. Acest pointer permite accesul la elementul respectiv. Funcţia se apelează simplu printr-o expresie de forma:

```
B *p = acceselem(indice);
```

Problema care apare în acest moment este aceea a utilizării pointerului *p*. Acesta este un pointer spre un obiect al uneia din clasele ierarhiei. De exemplu, dacă *p* are ca valoare adresa unui obiect al clasei Ai, atunci vom utiliza pointerul respectiv convertindu-l în mod explicit ca pointer spre Ai:

```
(Ai *)p
```

Presupunem că funcţia *f* este o funcţie membru nestatică definită în clasa B şi supraîncărcată în toate clasele Ai ale ierarhiei şi cu aceeaşi listă de parametri. Atunci:

```
(Ai *)p -> f(...)
```

apelează funcţia membru nestatică a clasei Ai, pentru obiectul din container de adresă (Ai *)p.

Evident, dacă valoarea lui *p* returnată de funcţia *acceselem* nu este adresa unui obiect al clasei Ai, atunci conversia:

```
(Ai *)p
```

este eronată şi apelul:

```
(Ai *)p -> f(...)
```

va fi şi el eronat.

De aceea, accesele de acest fel la obiectele unui container sînt surse de erori care nu pot fi depistate decît destul de greu de către programator.

Programatorul trebuie să ţină evidenţa tipurilor obiectelor păstrate în

container şi pe baza acesteia să realizeze conversia de pointeri corespunzătoare.

Ar fi mult mai convenabil să putem apela funcţiile *f* fără a mai face conversia explicită a lui *p* spre tipul Ai.

Cu alte cuvinte, un apel de forma:

```
p -> f(...)
```

să apeleze funcţia membru nestatică *f* a clasei Ai, dacă *p* are ca valoare adresa unui obiect al clasei Ai. Aceasta l-ar scuti pe programator de a mai ţine evidenţa tipurilor obiectelor puse în container şi de a face conversii de felul (Ai *).

Apelul:

```
p -> f(...)
```

poate fi tratat într-un singur mod la compilare şi anume se apelează funcţia membru *f* a clasei de bază B. Aceasta, deoarece *p* este pointer spre obiecte ale clasei B. Selectarea funcţiei *f* în funcţie de valoarea pointerului *p* se poate realiza numai la execuţie. De aceea, pentru ca la apelul de forma:

```
p -> f(...)
```

să se selecteze funcţia membru *f* a clasei Ai cînd *p* are ca valoare adresa unui obiect al clasei Ai, este necesar ca apelul respectiv să nu fie rezolvat la compilare, ci la execuţie. În acest caz, legătura dintre obiect şi funcţia membru apelată nu mai este statică ci *dinamică*. Se obişnuieşte să se mai spună că legătura respectivă este *întîrziată* sau *ulterioară* (late binding).

Legăturile de acest fel se realizează cu ajutorul funcţiilor *virtuale*. O funcţie *virtuală* este o funcţie membru nestatică a cărei prototip sau antet este precedat de cuvîntul cheie *virtual*. O astfel de funcţie se defineşte ca fiind virtuală în clasa de bază şi apoi ea poate fi definită în clasele derivate pentru care este nevoie să se utilizeze legătura dinamică. În clasele derivate nu este nevoie să se mai utilizeze cuvîntul cheie *virtual* în prototipul sau antetul funcţiilor virtuale. Deci, dacă funcţia *f*, din exemplul de mai sus, este declarată ca virtuală în clasa de bază B:

```
virtual ... f(...);
```

atunci funcţia *f* definită cu aceeaşi listă de parametri este în mod automat virtuală pentru orice clasă Ai derivată direct sau indirect din B.

Funcţia *f* apare ca o funcţie care are mai multe versiuni, una corespunde clasei de bază, iar celelalte claselor derivate.

Se spune că funcţiile virtuale din clasele derivate se suprapun (în engleză

override) peste funcţiile virtuale din clasele lor de bază. Astfel, dacă B este o clasă de bază şi f o funcţie membru virtuală a acesteia, iar A este o clasă derivată din B pentru care *f* este redefinită ca funcţie membru cu aceeaşi listă de parametri, atunci vom spune că funcţia membru *f* a lui A se suprapune peste funcţia membru *f* a lui B.

Nu este obligatoriu ca fiecare clasă derivată să aibă o funcţie membru care să se suprapună peste funcţia virtuală definită în clasa de bază directă sau indirectă. Dacă o clasă derivată nu are o astfel de funcţie membru, atunci ea moşteneşte funcţia membru virtuală corespunzătoare primei clase din ierarhie de pe un nivel precedent şi care are definită funcţia respectivă.

Selectarea versiunii care să se apeleze se face pe baza unor tabele suplimentare construite în program în mod automat pentru funcţiile virtuale. De asemenea, obiectele claselor care conţin funcţii virtuale conţin adresele tabelelor care le corespund. De aceea, utilizarea funcţiilor virtuale necesită atît memorie suplimentară, cît şi un anumit timp suplimentar la execuţia programului, ambele fiind necesare pentru selectarea versiunilor funcţiilor virtuale.

Rezultă că funcţiile virtuale, deşi sînt avantajoase în multe cazuri, este bine să nu se facă exagerări în utilizarea lor deoarece conduc la un consum sporit de memorie şi timp de execuţie.

În legatură cu funcţiile virtuale este necesar să ţinem seama de următoarele reguli:

1. Funcţiile virtuale sînt funcţii membru nestatice.
2. Constructorii nu pot fi funcţii virtuale.
3. Destructorii pot fi funcţii virtuale.
4. Funcţiile inline nu pot fi funcţii virtuale.

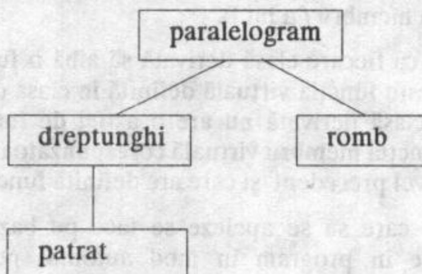
Exerciţii:

30.1 Să se implementeze o ierarhie de clase pentru următoarele patrulatere: paralelogram, dreptunghi, pătrat şi romb.

Obiectele acestor clase sînt patrulatere care au două laturi paralele cu axa *Ox*. Clasele au funcţii membru care permit realizarea următoarelor operaţii asupra obiectelor:

- afişare pe ecranul grafic;
- ştergere de pe ecranul grafic;
- deplasare pe ecranul grafic;
- calculul perimetrului şi ariei figurii geometrice.

Paralelogramul se află în rădăcina ierarhiei şi va fi clasa de bază pentru dreptunghi şi romb. Pătratul este o clasă derivată din dreptunghi. În felul acesta se obţine ierarhia din figura de mai jos:



Figurile se definesc cu ajutorul coordonatelor vîrfurilor. În cazul paralelogramului şi rombului se dau coordonatele la toate cele 4 vîrfuri. În cazul dreptunghiului şi pătratului se dau coordonatele a două vîrfuri diagonal opuse.

Constructorii claselor *paralelogram* şi *romb* verifică dacă laturile opuse sînt paralele şi dacă segmentul format cu primele două vîrfuri este paralel cu axa *Ox*.

Clasa *paralelogram* are o dată membru numită *valid*, care are valoarea 1 dacă obiectul curent este un paralelogram şi zero în caz contrar.

În mod analog, şi celelalte clase au date membru care stabilesc validitatea obiectului curent.

Alte date membru ale clasei *paralelogram* definesc caracteristicile pentru afişarea figurilor grafice:

- culoarea liniei;

şi

- grosimea liniei.

FIŞIERUL BXXX1A.H

```
#ifndef __GRAPHICS_H
#include <graphics.h>
#define __GRAPHICS_H
#endif
#ifndef __GSEGMENT_H
#include "BXXIX11.CPP"
#define __GSEGMENT_H
```

```
#endif

class paralelogram { //clasa radacina a ierarhiei
 protected:
    punct tvirf[4];
    int valid;
    int pvizibil;
    int culoare_latura;
    int grosime_latura;
 public:
    int  validparalel();
    /* valid = 1 daca obiectul curent este un paralelogram cu 2
                        laturi paralele cu Ox;
       valid = 0, altfel.
    */
    paralelogram();   //constructor implicit
    paralelogram(double xa,double ya,double xb,
                 double yb,double xc,double yc,
                 double xd,double yd,
                  int c=WHITE,int g=NORM_WIDTH);
    //constructor utilizat la initializari

    paralelogram(const punct& a,const punct& b,
                  const punct &c,const punct& d,
                  int cl=WHITE,int g=NORM_WIDTH);
    //constructor utilizat la initializari

    paralelogram(const paralelogram&);
    //constructor de copiere

    paralelogram& operator = (const
                                    paralelogram&);
    //supraincarca operatorul =

    virtual void afiseaza();
    //afiseaza figura curenta

    virtual void sterge();
    //face invizibila figura curenta

    virtual double perimetru() const;
    //returneaza perimetrul figurii curente
```

```
    virtual double arie() const;
    // returneaza aria figurii curente

    void deplaseaza(double dx,double dy);
    /* deplaseaza figura curenta modificind coordonatele:
       abscisa cu dx si ordonata cu dy */

};
```

FIŞIERUL BXXX1B.H

```
#ifndef __PARALELOGRAM.
#include "BXXX1A.H"
#define __PARALELOGRAM
#endif

class romb:public paralelogram {
    int rvalid;
 public:
    int validromb();
    /* rvalid=1, daca obiectul curent este romb si are doua laturi
                   paralele cu Ox;
       rvalid=0, altfel. */
    romb();    // constructor implicit
    romb(double xa,double ya,double xb,double yb,
        double xc,double yc,double xd,double yd,
        int c=WHITE,int g=NORM_WIDTH);
    // constructor pentru initializari

    romb(const punct& a,const punct& b,
        const punct& c,const punct& d,
        int cl=WHITE,int g=NORM_WIDTH);
    // constructor pentru initializari

    romb(const romb&);
    // constructor de copiere

    romb& operator = (const romb&);
    // supraincarca operatorul =

    /* - se mostenesc de la clasa paralelogram functiile virtuale:
             afiseaza si sterge
       - se mosteneste functia membru deplaseaza, nevirtuala.
    */
```

```
    double perimetru() const;
  /* - returneaza perimetrul rombului curent;
     - este functie virtuala deoarece se suprapune peste functia
       perimetru din clasa de baza care este functie virtuala. */

    double arie() const;
  /* - returneaza aria rombului curent;
     - este functie virtuala deoarece se suprapune peste functia
       arie din clasa de baza care este functie virtuala. */

};
```

FIŞIERUL BXXX1C.H

```
#ifndef __PARALELOGRAM
#include "BXXX1A.H"
#define __PARALELOGRAM
#endif

class dreptunghi:public paralelogram {
 protected:
    int dvalid;
 public:
    int validdrept();
    /* dvalid=1, daca obiectul curent este un
                  dreptunghi si 2 laturi sint paralele cu Ox;
       dvalid=0, altfel.
    */

    dreptunghi(); //constructor implicit
    dreptunghi(double xa,double ya,double xc,
               double yc,int c=WHITE,
               int g=NORM_WIDTH);
    // constructor utilizat la initializari

    dreptunghi(const punct& a,const punct& c,
               int cl=WHITE,int g=NORM_WIDTH);
    // constructor utilizat la initializari

    dreptunghi (const dreptunghi&);
    // constructor de copiere

    dreptunghi& operator = (const dreptunghi&);
    // supraincarca operatorul =
```

```
    void afiseaza();
    /* functie virtuala pentru afisarea dreptunghiului curent */

    void sterge();
    /* functie virtuala pentru a face invizibil dreptunghiul curent */

    double perimetru() const;
    /* functie virtuala care returneaza perimetrul figurii curente */

    double arie() const;
    /* functie virtuala care returneaza aria figurii curente */

    // se mosteneste functia obisnuita deplaseaza
};
```

FIŞIERUL BXXX1D.H

```
#ifndef __DREPTUNGHI
#include "BXXX1C.H"
#define __DREPTUNGHI
#endif

class patrat:public dreptunghi {
 protected:
    int pvalid;
 public:

    int validpatrat();
    /* pvalid=1, daca obiectul curent este un patrat si
                     are 2 laturi paralele cu Ox;
       pvalid=0, altfel. */

    patrat();   // constructor implicit
    patrat(double xa,double ya,double xc,
           double yc,int c=WHITE,
           int g=NORM_WIDTH);
    // constructor utilizat la initializari

    patrat(const punct& a,const punct& c,
          int cl=WHITE,int g=NORM_WIDTH);
    // constructor utilizat la initializari

    patrat(const patrat&);
    // constructor de copiere
```

```
    patrat& operator = (const patrat&);
    // supraincarca operatorul =

    /* - se mostenesc functiile virtuale:
              afiseaza si sterge
         de la clasa dreptunghi
       - se mosteneste functia membru deplasare de la paralelogram
    */

    double perimetru() const;
   /* functie virtuala care returneaza perimetrul obiectului curent */

    double arie() const;
   /* functie virtuala care returneaza aria obiectului curent */
};
```

Funcţiile membru ale ierarhiei de clase se definesc în fişierele de mai jos, de extensie *CPP*.

FIŞIERUL BXXX1A

```
#ifndef __PARALELOGRAM
#include "BXXX1A.H"
#define __PARALELOGRAM
#endif

// clasa PARALELOGRAM
int paralelogram::validparalel()
/* valid=1, daca obiectul curent are laturile opuse paralele si 2 laturi sint
   paralele cu axa Ox */
{
 gsegment OX(1,0,0,0);   // axa OX
 // se definesc laturile paralelogramului

 punct A(tvirf[0]);
 punct B(tvirf[1]);
 punct C(tvirf[2]);
 punct D(tvirf[3]);

 gsegment AB(tvirf[0],tvirf[1]);
 gsegment BC(tvirf[1],tvirf[2]);
 gsegment CD(tvirf[2],tvirf[3]);
 gsegment DA(tvirf[3],tvirf[0]);

 valid = 0;
```

```
 if(AB || CD)          //segmentele AB si CD sint paralele
    if(AB || OX)       //segmentele AB si CD sint paralele cu OX
       if(BC || DA) //segmentele BC si DA sint paralele
          valid = 1;

 // test daca punctele sint diferite
 if(A == B)
    valid = 0;
 if(A == C)
    valid = 0;
 if(A == D)
    valid = 0;
 if(B == C)
    valid = 0;
 if(B == D)
    valid = 0;
 if(C == D)
    valid = 0;
 return valid;
}

inline paralelogram::paralelogram() // constructor implicit
{
 valid = 0;
 pvizibil=0;
}
paralelogram::paralelogram(double xa,double ya,
                           double xb,double yb,
                           double xc,double yc,
                           double xd,double yd,
                           int c,int g)
// constructor pentru initializari
{
 tvirf[0] = punct(xa,ya);
 tvirf[1] = punct(xb,yb);
 tvirf[2] = punct(xc,yc);
 tvirf[3] = punct(xd,yd);

 // se valideaza paralelogramul
 validparalel();
 pvizibil = 0;
 culoare_latura = c;
```

```
 grosime_latura = g;
}

paralelogram::paralelogram(const punct& a,
                        const punct& b,const punct& c,
                        const punct& d,int cl,int g)
// constructor folosit la initializari
{
 tvirf[0] = a;
 tvirf[1] = b;
 tvirf[2] = c;
 tvirf[3] = d;

 // se valideaza paralelogramul
 validparalel();
 pvizibil = 0;
 culoare_latura = cl;
 grosime_latura = g;
}

paralelogram::paralelogram (const paralelogram& p)
// constructor de copiere
{
 for(int i=0;i < 4;i++)
    tvirf[i]=p.tvirf[i];
 valid = p.valid;
 pvizibil=p.pvizibil;
 culoare_latura = p.culoare_latura;
 grosime_latura = p.grosime_latura;
}

paralelogram& paralelogram::operator = (const
                              paralelogram& p)
// supraincarca operatorul =
{
 if(p.valid && this != &p){
    for(int i=0;i < 4;i++)
       tvirf[i]=p.tvirf[i];
    valid=p.valid;
    pvizibil=p.pvizibil;
    culoare_latura=p.culoare_latura;
    grosime_latura=p.grosime_latura;
 }
```

```
 else
    if(p.valid == 0)
        valid = 0;
 return *this;
}

void paralelogram::afiseaza()
// functie virtuala care afiseaza paralelogramul curent
{
 if(valid){     // paralelogramul este valid
    gsegment AB(tvirf[0],tvirf[1],culoare_latura,
                grosime_latura);
    AB.afiseaza();  // afiseaza segmentul AB
    gsegment BC(tvirf[1],tvirf[2],culoare_latura,
                grosime_latura);
    BC.afiseaza();  // afiseaza segmentul BC
    gsegment CD(tvirf[2],tvirf[3],culoare_latura,
                grosime_latura);
    CD.afiseaza();  // afiseaza segmentul CD
    gsegment DA(tvirf[3],tvirf[0],culoare_latura,
               grosime_latura);
    DA.afiseaza();  // afiseaza segmentul DA
    pvizibil=1;
 }
}

void paralelogram::sterge()
// functie virtuala care face invizibil paralelogramul curent
{
 if(valid){     // paralelogramul este valid
    gsegment AB(tvirf[0],tvirf[1],culoare_latura,
                grosime_latura);
    AB.sterge();
    gsegment BC(tvirf[1],tvirf[2],culoare_latura,
                grosime_latura);
    BC.sterge();
    gsegment CD(tvirf[2],tvirf[3],culoare_latura,
                grosime_latura);
    CD.sterge();
    gsegment DA(tvirf[3],tvirf[0],culoare_latura,
                grosime_latura);
    DA.sterge();
    pvizibil=0;
```

```
  }
}

double paralelogram::perimetru() const
/* functie virtuala care returneaza perimetrul paralelogramului curent */
{
 if(valid) {
     double ab = tvirf[0].dist(tvirf[1]);
     double bc = tvirf[1].dist(tvirf[2]);
     return 2*ab+2*bc;
 }
 return 0;
}

double paralelogram::arie() const
/* functie virtuala care returneaza aria paralelogramului curent */
{
 if(valid){
     gsegment CD(tvirf[2],tvirf[3]);
     gpunct A=gpunct(tvirf[0],culoare_latura);

     // se determina inaltimea paralelogramului
     gsegment I = CD|A;
     punct E=I*CD;   // intersectia inaltimii din A cu latura CD
     double cd=tvirf[2].dist(tvirf[3]);
                                        // lungimea bazei
     double ae=E.dist(A);      // lungimea inaltimei
     return cd*ae;
 }
 return 0;
}

void paralelogram::deplaseaza(double dx,double dy)
/* deplaseaza paralelogramul modificind abscisa punctelor cu dx si
   ordonata cu dy */
{
 if(valid) {
     sterge();  // modifica virfurile figurii
     for(int i=0;i < 4;i++)
         tvirf[i] = tvirf[i]+punct(dx,dy);
     afiseaza();
 }
}
```

FIŞIERUL BXXX1B

```
#ifndef __PARALELOGRAM_H
#include "BXXX1A.CPP"
#define __PARALELOGRAM_H
#endif
#ifndef __ROMB_H
#include "BXXX1B.H"
#define __ROMB_H
#endif

// clasa ROMB
int romb::validromb()
/* rvalid=1, daca obiectul curent este un romb si are doua laturi paralele
              cu axa Ox;
   rvalid=0, altfel.
*/
{
 // se verifica daca figura este paralelogram

 validparalel();

 if(valid) {

     // se verifica daca 2 laturi alaturate sint egale
     double ab=tvirf[0].dist(tvirf[1]);
     double bc=tvirf[1].dist(tvirf[2]);

     if(testegal(ab,bc))
         rvalid = 1;   // laturi egale
     else
         rvalid = 0;   // nu este romb
 }
 else
         rvalid = 0;    // nu este paralelogram
 return rvalid;
}

inline romb::romb()  // constructor implicit
{
 rvalid = 0;
}
```

```
inline romb::romb(double xa,double ya,
          double xb,double yb,double xc,
          double yc,double xd,double yd,
           int c,int g):
      paralelogram(xa,ya,xb,yb,xc,yc,xd,yd,c,g)
// constructor utilizat la initializari
{
 if(valid)     // figura este paralelogram
    validromb();   /* rvalid=1 daca figura este romb;
                      altfel rvalid=0 */
 else
    rvalid=0;      /* figura nu este un paralelogram si deci
                      nu este nici romb */
}

inline romb::romb(const punct& a,const punct& b,
                  const punct& c,const punct& d,
                  int cl,int g):
     paralelogram(a,b,c,d,cl,g)
// constructor utilizat la initializari
{
 if(valid)
    validromb();
 else
    rvalid=0;
}

inline romb::romb(const romb& r):paralelogram(r)
// constructor de copiere
{
 rvalid=r.rvalid;
}

romb& romb::operator = (const romb& r)
// supraincarca operatorul =
{
 if(r.rvalid && this != &r) {
    for(int i=0;i < 4;i++)
       tvirf[i] = r.tvirf[i];
    valid = r.valid;
    culoare_latura = r.culoare_latura;
    grosime_latura = r.grosime_latura;
    pvizibil = r.pvizibil;
```

```
    rvalid = r.rvalid;
  }
  else
    if(r.rvalid == 0)
      rvalid = 0;
  return *this;
}

double romb::perimetru() const
// returneaza perimetrul rombului curent
{
  if(rvalid) {
    double p=4*tvirf[0].dist(tvirf[1]);
    return p;
  }
  return 0;   // figura nu este romb
}

double romb::arie() const  // returneaza aria rombului curent
{
  if(rvalid){
    double d1=tvirf[0].dist(tvirf[2]);
    double d2=tvirf[1].dist(tvirf[3]);
    return d1*d2/2;
  }
  return 0;   // figura nu este romb
}
```

FIŞIERUL BXXX1C

```
#ifndef __PARALELOGRAM
#include "BXXX1A.CPP"
#define __PARALELOGRAM
#endif
#ifndef __DREPTUNGHI
#include "BXXX1C.H"
#define __DREPTUNGHI
#endif

// clasa DREPTUNGHI
int dreptunghi::validdrept()
/* dvalid=1, daca obiectul curent este un dreptunghi si are doua laturi
             opuse paralele cu axa Ox;
   dvalid=0, altfel. */
```

```
{
 // se verifica daca figura este paralelogram cu 2 laturi paralele cu axa Ox
 validparalel();
 dvalid = 0;

 if(valid){

     // se verifica daca 2 laturi alaturate sint perpendiculare
     gsegment AB(tvirf[0],tvirf[1]);
     gsegment BC(tvirf[1],tvirf[2]);
     if(AB && BC)
         dvalid = 1;  // figura este dreptunghi
 }
 return dvalid;
}

inline dreptunghi::dreptunghi()  // constructor implicit
{
 dvalid = 0;
}

inline dreptunghi::dreptunghi(double xa,double ya,
                double xc,double yc,
                int c,int g):
      paralelogram(xa,ya,xc,ya,xc,yc,xa,yc,c,g)
// constructor utilizat la initializari
{
 if(valid)
    validdrept();
 else
    dvalid=0;
}

dreptunghi::dreptunghi(const punct& a,
                    const punct& c,int cl,int g)
// constructor utilizat la initializari
{
 tvirf[0] = a;
 tvirf[1] = punct(c.retabs(),a.retord());
 tvirf[2] = c;
 tvirf[3] = punct(a.retabs(),c.retord());

 validdrept();
```

```
 pvizibil = 0;
 culoare_latura = cl;
 grosime_latura = g;
}

dreptunghi::dreptunghi (const dreptunghi& d)
// constructor de copiere
{
 for(int i=0;i < 4;i++)
    tvirf[i]=d.tvirf[i];
 valid = d.valid;
 dvalid = d.dvalid;
 pvizibil = d.pvizibil;
 culoare_latura = d.culoare_latura;
 grosime_latura = d.grosime_latura;
}

dreptunghi& dreptunghi::operator = (
                      const dreptunghi& d)
// supraincarca operatorul =
{
 if(d.dvalid && this != &d){
    for(int i=0;i < 4;i++)
       tvirf[i]=d.tvirf[i];
    valid=d.valid;
    dvalid=d.dvalid;
    pvizibil=d.pvizibil;
    culoare_latura=d.culoare_latura;
    grosime_latura=d.grosime_latura;
 }
 else
    if(d.dvalid == 0)
       dvalid = 0;
 return *this;
}

void dreptunghi::afiseaza()
// afiseaza dreptunghiul curent
{
 if(dvalid){     // dreptunghi valid
    struct linesettingstype temp;
    getlinesettings(&temp);
    int c = getcolor();
```

```
    setlinestyle(SOLID_LINE,0,grosime_latura);
    setcolor(culoare_latura);
    rectangle(tvirf[0].retabs(),tvirf[0].retord(),
          tvirf[2].retabs(),tvirf[2].retord());
    setlinestyle(temp.linestyle,
                temp.upattern,temp.thickness);
    setcolor(c);
    pvizibil =1;
 }
}

void dreptunghi::sterge()
// functie virtuala care face invizibil dreptunghiul curent
{
 if(dvalid){
    int temp = culoare_latura;
    culoare_latura = getbkcolor();
    afiseaza();
    culoare_latura=temp;
    pvizibil=0;
 }
}

double dreptunghi::perimetru() const
/* functie virtuala care returneaza perimetrul dreptunghiului curent */
{
 if(dvalid) {
    double baza = tvirf[1].retabs()-
                  tvirf[0].retabs();
    double inaltime = tvirf[3].retord()-
                      tvirf[0].retord();
    if(baza < 0)
       baza = -baza;
    if(inaltime < 0)
       inaltime = -inaltime;
    return 2*(baza+inaltime);
 }
 return 0;
}
```

```
double dreptunghi::arie() const
/* functie virtuala care returneaza aria dreptunghiului curent */
{
 if(dvalid){
     double baza=tvirf[1].retabs()-
                 tvirf[0].retabs();
     double inaltime=tvirf[3].retord()-
                     tvirf[0].retord();
     if(baza < 0)
        baza = -baza;
     if(inaltime < 0)
        inaltime = -inaltime;
     return baza*inaltime;
 }
 return 0;
}
```

FIŞIERUL BXXX1D

```
#ifndef __DREPTUNGHI_H
#include "BXXX1C.CPP"
#define __DREPTUNGHI_H
#endif
#ifndef __PATRAT
#include "BXXX1D.H"
#define __PATRAT
#endif

int patrat::validpatrat()
/* - pvalid=1 daca obiectul curent este un patrat;
   - pvalid=0 altfel. */
{
 validdrept();
 pvalid=0;
 if(dvalid) {
 //verifica daca 2 laturi alaturate sint egale

     double ab=tvirf[1].retabs()-tvirf[0].retabs();
     double da=tvirf[3].retord()-tvirf[0].retord();
     if(ab < 0)
        ab=-ab;
     if(da < 0)
        da = -da;
     if(testegal(ab,da))
```

```
        pvalid=1;  //figura este patrat
  }
  return pvalid;
 }

inline patrat::patrat()  //constructor implicit
{
 pvalid=0;
}

inline patrat::patrat(double xa,double ya,
                       double xc,double yc,
                       int c,int g):
        dreptunghi(xa,ya,xc,yc,c,g)
//constructor pentru initializari
{
 if(dvalid)
    validpatrat();
 else
    pvalid=0;
}

inline patrat::patrat(const punct& a,
                     const punct&c,int cl,int g):
        dreptunghi(a,c,cl,g)
//constructor utilizat la initializari
{
 if(dvalid)
    validpatrat();
 else
    pvalid=0;
}

inline patrat::patrat(const patrat& p):
        dreptunghi(p)
//constructor de copiere
{
 pvalid=p.pvalid;
}
```

```
patrat& patrat::operator = (const patrat& p)
// supraincarca operatorul =
{
 if(p.pvalid && this != &p){
    for(int i=0;i < 4;i++)
       tvirf[i] = p.tvirf[i];
    valid=p.valid;
    dvalid=p.dvalid;
    pvalid=p.pvalid;
    culoare_latura=p.culoare_latura;
    grosime_latura=p.grosime_latura;
    pvizibil=p.pvizibil;
 }
 else
    if(p.pvalid==0)
       pvalid=0;
 return *this;
}

double patrat::perimetru() const
// returneaza perimetrul patratului curent
{
 if(pvalid){
    double latura=tvirf[1].retabs()-
                  tvirf[0].retabs();
    if(latura < 0)
       latura = -latura;
    return 4*latura;
 }
 else
    return 0;
}

double patrat::arie() const
// returneaza aria patratului curent
{
 if(pvalid){
    double latura=tvirf[1].retabs()-
                  tvirf[0].retabs();
    return latura*latura;
 }
 else
    return 0;
```

}

30.2 Să se scrie un program care afişează în jumătatea de sus a ecranului următoarele patrulatere:

- un paralelogram;
- un romb;
- un dreptunghi;
- un pătrat.

După afişare se acţionează o tastă ASCII oarecare şi figurile respective se afişează în jumătatea de jos a ecranului. Acţionînd din nou o tastă ASCII, se va afişa, pentru fiecare figură geometrică, perimetrul şi aria ei.

PROGRAMUL BXXX2

```
#include <conio.h>
#include "BXXX1B.CPP"
#include "BXXX1D.CPP"

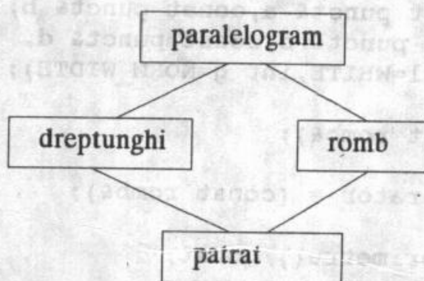
main()
/* - afiseaza urmatoarele patrulatere:
        - un paralelogram;
        - un romb;
        - un dreptunghi;
        - un patrat,
      in jumatatea de sus a ecranului;
   - dupa actionarea unei taste ASCII, patrulaterele respective se
     afiseaza in jumatatea de jos a ecranului;
   - actionind din nou o tasta ASCII se va afisa perimetrul si aria
      fiecarui patrulater.
*/
{
 paralelogram par(50,30,100,30,70,100,20,100);
 romb r(130,30,180,30,150,70,100,70,YELLOW);
 dreptunghi d(200,30,300,150,LIGHTRED);
 patrat p(350,30,450,130,LIGHTBLUE);
 int gd=DETECT,gm;
 initgraph(&gd,&gm,"C:\\borlandc\\bgi");
 par.afiseaza();
 r.afiseaza();
 d.afiseaza();
 p.afiseaza();
 getch();
```

```
    par.deplaseaza(10,170);
    r.deplaseaza(10,170);
    d.deplaseaza(10,170);
    p.deplaseaza(10,170);
    getch();
    closegraph();
    double ppar = par.perimetru();
    double pr = r.perimetru();
    double pd = d.perimetru();
    double pp = p.perimetru();
    printf("\tperimetru\n");
    printf("paralelogram = %g\n",ppar);
    printf("romb = %g\n",pr);
    printf("dreptunghi = %g\n",pd);
    printf("patrat = %g\n",pp);
    double apar = par.arie();
    double ar = r.arie();
    double ad = d.arie();
    double ap = p.arie();
    printf("\taria\n");
    printf("paralelogram = %g\n",apar);
    printf("romb = %g\n",ar);
    printf("dreptunghi = %g\n",ad);
    printf("patrat = %g\n",ap);
    getch();
}
```

30.3 În exercițiul 30.1, s-a implementat o ierarhie de patrulatere în vîrful căreia se află clasa *paralelogram*. Clasele *dreptunghi* şi *romb* sînt clase derivate din clasa *paralelogram*. Clasa *patrat* este o clasă derivată din clasa *dreptunghi*.

În exercițiul de față se implementează aceeaşi ierarhie de clase cu singura deosebire că clasa *patrat* este derivată atît din clasa *dreptunghi* cît şi din clasa *romb*. Pentru a nu moşteni clasa *paralelogram* de două ori, clasa *paralelogram* se defineşte ca o clasă de bază virtuală pentru clasele *dreptunghi* şi *romb*.

Ierarhia de clase implementată în acest exercițiu este schematizată mai jos:



Pentru a implementa o ierarhie de patrulatere sub forma indicată mai sus, se fac următoarele modificări:

- în fişierele de extensie H, care conţin definiţiile pentru clasele *dreptunghi* şi *romb*, la derivarea lor, se utilizează clasa de bază virtuală *paralelogram*:

```
class dreptunghi::virtual public
  paralelogram {
   ...
};
class romb::virtual public paralelogram {
   ...
};
```

- se modifică clasa *patrat*, aceasta devenind o clasă derivată atît din clasa *dreptunghi* cît şi din clasa *romb*.

FIŞIERUL BXXX3B.H

```
#ifndef __PARALELOGRAM
#include "BXXX1A.H"
#define __PARALELOGRAM
#endif

class romb:virtual public paralelogram {
 protected:
    int rvalid;
 public:
    int validromb();
    romb();  // constructor implicit
    romb(double xa,double ya,double xb,double yb,
         double xc,double yc,double xd,double yd,
         int c=WHITE,int g=NORM_WIDTH);
```

```
    romb(const punct& a,const punct& b,
         const punct& c,const punct& d,
         int cl=WHITE,int g=NORM_WIDTH);

    romb(const romb&);

    romb& operator = (const romb&);

    double perimetru() const;

    double arie() const;
};
```

FIŞIERUL BXXX3C.H

```
#ifndef __PARALELOGRAM
#include "BXXX1A.H"
#define __PARALELOGRAM
#endif

class dreptunghi:virtual public paralelogram {
 protected:
    int dvalid;
 public:
    int validdrept();
    dreptunghi();
    dreptunghi(double xa,double ya,
               double xc,double yc,
               int c=WHITE,int g=NORM_WIDTH);

    dreptunghi(const punct& a,const punct& c,
               int cl=WHITE,int g=NORM_WIDTH);

    dreptunghi (const dreptunghi&);

    dreptunghi& operator = (const dreptunghi&);

    void afiseaza();

    void sterge();

    double perimetru() const;

    double arie() const;
```

```
};
```

FIŞIERUL BXXX3D.H

```
#ifndef __DREPTUNGHI
#include "BXXX3C.H"
#define __DREPTUNGHI
#endif
#ifndef __ROMB
#include "BXXX3B.H"
#endif

class patrat:virtual public dreptunghi,
        virtual public romb {
 protected:
    int pvalid;
 public:
    int validpatrat();
    patrat();   //constructor implicit
    patrat(double xa,double ya,double xc,
        double yc,
    int c=WHITE,int g=NORM_WIDTH);

    patrat(const punct& a,const punct& c,
        int cl=WHITE,int g=NORM_WIDTH);

    patrat(const patrat&);

    patrat& operator = (const patrat&);

    double perimetru() const;

    double arie() const;
};
```

Funcţiile membru pentru clasele *paralelogram*, *romb* şi *dreptunghi* sînt aceleaşi ca în cazul exerciţiului 30.1, deci ele sînt definite în fişierele:

- BXXX1A.CPP - pentru paralelogram;
- BXXX1B.CPP - pentru romb;
- BXXX1C.CPP - pentru dreptunghi.

Mai jos, se definesc funcţiile membru noi pentru clasa *patrat*.

FIŞIERUL BXXX3

```
#ifndef __ROMB
#include "BXXX3B.H"      // clasa romb
#define __ROMB
#endif
#ifndef __DREPTUNGHI
#include "BXXX3C.H"      // clasa dreptunghi
#define __DREPTUNGHI
#endif
#ifndef __PATRAT
#include "BXXX3D.H"      // clasa patrat
#define __PATRAT
#endif
#ifndef __ROMB_H
#define __ROMB_H
#include "BXXX1B.CPP"  // functii membru pentru clasa romb
#endif
#ifndef __DREPTUNGHI_H
#include "BXXX1C.CPP"  // functii membru pentru
                       // clasa dreptunghi
#define __DREPTUNGHI_H
#endif

int patrat::validpatrat()
/* - pvalid=1, daca obiectul curent este patrat (este atit dreptunghi cit si
             romb);
   - pvalid = 0, altfel.
*/
{
 validdrept();/* stabileste daca obiectul curent este dreptunghi */
 validromb();     // stabileste daca obiectul curent este romb
 pvalid = rvalid & dvalid;
 return pvalid;
}

inline patrat::patrat()  // constructor implicit
{
 pvalid = 0;
}
```

```
inline patrat::patrat(double xa,double ya,
            double xc,double yc,int c,int g):
        dreptunghi(xa,ya,xc,yc,c,g)
```
// constructor pentru initializari
```
{
 if(dvalid)
    validpatrat();
 else
    pvalid=0;
}

inline patrat::patrat(const punct& a,
            const punct& c,int cl,int g):
        dreptunghi(a,c,cl,g)
```
// constructor utilizat pentru initializari
```
{
 if(dvalid)
    validpatrat();
 else
    pvalid=0;
}

inline patrat::patrat(const patrat& p):
        dreptunghi(p)
```
// constructor de copiere
```
{
 rvalid = p.rvalid;
 pvalid = p.pvalid;
}

patrat& patrat::operator = (const patrat& p)
```
// supraincarca operatorul de atribuire
```
{
 if(p.pvalid && this != &p){
    for(int i=0;i < 4;i++)
        tvirf[i] = p.tvirf[i];
    valid = p.valid;
    rvalid = p.rvalid;
    dvalid = p.dvalid;
    pvalid = p.pvalid;
    pvizibil = p.pvizibil;
    culoare_latura = p.culoare_latura;
    grosime_latura = p.grosime_latura;
```

```
 }
 else
     if(p.pvalid == 0)
         pvalid = 0;
 return *this;
}

inline double patrat::perimetru() const
// returneaza perimetrul patratului curent
{
 if(pvalid)
     return romb::perimetru();
     /* se apeleaza functia membru a clasei romb pentru
        calculul perimetrului patratului curent */
 return 0;
}

double patrat::arie() const
// returneaza aria patratului curent
{
 if(pvalid){
     double latura=tvirf[1].retabs()-
                   tvirf[0].retabs();
     return latura*latura;
 }else
     return 0;
}
```

30.4 Să se implementeze un container pentru obiecte ale ierarhiei de clase definită în exercițiul 30.3.

În exercițiul de față, containerul se implementează printr- un tablou de pointeri spre obiecte de tip paralelogram.

Dimensiunea maximă a tabloului se ia egală cu 100.

FIŞIERUL BXXX4.H

```
#ifndef __PATRAT_H
#include "BXXX3.CPP"
#define __PATRAT_H
#endif
```

```
class container {
    paralelogram *tcontainer[100];
    int n;    // indicele elementului liber
 public:
    container();  // constructor implicit
    int atrib (paralelogram *p);
    // pune un obiect in container

    paralelogram *acceselem(int i);
    // permite acces la obiectul spre care pointeaza tcontainer [i]
};
```

Funcţiile membru se definesc în fişierul de mai jos,de extensie *CPP*.

FIŞIERUL BXXX4

```
#ifndef __CONTAINER_H
#include "BXXX4.H"
#define __CONTAINER_H
#endif

container::container()  // constructor implicit
{
 n=0;  // container vid
}

int container::atrib (paralelogram *p)
/* - pune obiectul curent in container daca acesta nu este plin;
   - returneaza 1 daca containerul nu este plin;
   - returneaza 0 altfel.
*/
{
 if(n < 100){
    tcontainer[n++] = p;
    return 1;
 }else
    return 0;  // container plin
}

paralelogram *container::acceselem(int i)
/* permite accesul la obiectul spre care pointeaza tcontainer[i] */
{
 if(0 <= i && i < n)
    return tcontainer[i];
```

```
else
    return 0;
}
```

30.5 Să se scrie o funcţie care stabileşte dacă un obiect de tip *punct* are coordonatele în limitele ecranului.

Funcţia returnează 1 dacă coordonatele obiectului parametru aparţin ecranului şi zero în caz contrar.

Tipul *punct* este definit în exerciţiile 29.1 şi 29.7.

FUNCŢIA BXXX5

```
#ifndef __GRAPHICS_H
#include <graphics.h>
#define __GRAPHICS_H
#endif
#ifndef __PUNCT_H
#include "BXXIX7.CPP"
#define __PUNCT_H
#endif

int ecran (const punct& p)
/* returneaza 1 daca coordonatele punctului p se afla in limitele
   ecranului si zero in caz contrar */
{
 double x=p.retabs();
 double y=p.retord();

 if(x < 0 || x > getmaxx())
     return 0;
 if(y < 0 || y > getmaxy())
     return 0;
 return 1;
}
```

30.6 Să se scrie un program care realizează următoarele:

a. Citeşte una din literele P,R,D,T (mică sau mare) care reprezintă tipul unui patrulater:

P - paralelogram;

R - romb;

D - dreptunghi;

T - pătrat.

b. Citeşte coordonatele vîrfurilor patrulaterului al cărui tip este precizat de litera citită la punctul *a*.

c. Păstrează, într-un container, patrulaterul ale cărui coordonate ale vîrfurilor au fost citite la punctul *b*, dacă acesta are următoarele proprietăţi:

- este unul din tipurile de patrulater amintite la punctul a;
- patrulaterul are 2 laturi paralele cu Ox;
- vîrfurile sînt situate în limitele ecranului grafic;

d. Punctele a-c se repetă pînă la întîlnirea sfîrşitului de fişier.

e. După întîlnirea sfîrşitului de fişier, pentru fiecare patrulater din container, se afişează pe rînd:

- figura pe care o reprezintă patrulaterul;
- perimetrul patrulaterului;
- aria figurii.

După afişarea datelor unei figuri, se va acţiona o tastă ASCII pentru a afişa datele figurii următoare din container.

Culoarea utilizată la afişarea patrulaterelor este definită de constanta simbolică YELLOW.

PROGRAMUL BXXX6

```
#include <conio.h>
#include "BXXX4.CPP" /* clasa container, precum si clasele
                        ierarhiei de patrulatere */
#include "BXXX5.CPP" // functia ecran

main()
/* - citeste una din literele P,R,D,T (mica sau mare) care reprezinta tipul
     unui patrulater:
        P - paralelogram;
        R - romb;
        D - dreptunghi;
        T - patrat;
   - citeste coordonatele virfurilor patrulaterului al carui tip a fost defi-
     nit de litera citita;
```

- pastreaza patrulaterul intr-un container daca:
    - este unul din tipurile de patrulater amintite mai sus;
    - are 2 laturi paralele cu Ox;
    - virfurile sint situate in limitele ecranului grafic;
- se repeta pasii de mai sus pina la intilnirea sfirsitului de fisier;
- la intilnirea sfirsitului de fisier, pentru fiecare patrulater din container, se afiseaza pe rind:
    - patrulaterul respectiv;
    - perimetrul si aria patrulaterului

```
*/
{
 char text1[]="virfurile tastate nu formeaza un\n";
 char text2[]="cu 2 laturi paralele cu Ox\n";
 char car[2],cr;
 int i;
 int gd=DETECT,gm;
 initgraph(&gd,&gm,"C:\\borlandc\\bgi");
 container patrulater;
 punct a,b,c,d;  // virfurile unui patrulater
 paralelogram *p;
 romb *r;
 dreptunghi *dr;
 patrat *t;

 for(;;) {      // citeste litera
     for(;;){
         printf("una din literele P, R, D,\
                 T sau <Ctrl>-Z\n");
         if((i=scanf("%1s",car)) == EOF)
             break;  // s-a intilnit sfirsitul de fisier
         cr=car[0];
         if(cr >= 'a')  // litera mica
             cr = cr-'a'+'A';
         if(cr=='P' || cr=='R' || cr=='D' ||
           cr=='T')
             break;
         printf("se cere una din literele:\n");
         printf("\tP\n\tR\n\tD\n\t\T\n");
         fflush(stdin);
     }  // sfirsit ciclu pentru citirea literei
     if(i == EOF)
         break;  // s-a intilnit sfirsitul de fisier
```

```
// se citesc virfurile patrulaterului si obiectul
// corespunzator se pune in container daca satisface
// conditiile cerute

fflush(stdin);
i = EOF;
switch(cr){
    case 'P':  // paralelogram
    case 'R':  // romb
        if(a.citpct("a") == 0)
            break;  // EOF
        if(b.citpct("b") == 0)
            break;  // EOF
        if(c.citpct("c") == 0)
            break;  // EOF
        if(d.citpct("d") == 0)
            break;  // EOF
        i=0;  // nu s-a intilnit sfirsitul de fisier
        if(ecran(a)==0 || ecran(b)==0 ||
          ecran(c)==0 || ecran(d)==0) {

            // cel putin un virf nu este in limitele ecranului grafic
            printf("cel putin un virf \
                    nu este pe ecran\n");
            break;    // iesire din switch
        }
        if(cr=='P'){   // paralelogram
            p=new paralelogram(a,b,c,d,YELLOW);
            if(p -> validparalel()==0){
                printf(text1);
                printf(" paralelogram ");
                printf(text2);
                break;  // iesire din switch
            }

            // se pune paralelogramul in container daca
            // exista loc
            if(patrulater.atrib(p) == 0)
                i = EOF;  // container plin
            break;      // iesire din switch
        }
        else{  // romb
```

```
            r=new romb(a,b,c,d,YELLOW);
            if(r->validromb()==0){
                printf(text1);
                printf(" romb ");
                printf(text2);
                break; // iesire din switch
            }
            // se pune rombul in container daca este loc
            if(patrulater.atrib(r)==0)
                i=EOF; // container plin
            break; // iesire din switch
        } // sfirsit else
    default:  // dreptunghi sau patrat
        if(a.citpct("a") == 0)
            break;  // EOF
        if(c.citpct("c") == 0)
            break;  // EOF
        i=0; // nu s-a intilnit sfirsitul de fisier
        if(ecran(a) == 0 || ecran(c) == 0){
        // cel putin un virf nu este in limitele ecranului grafic

        printf("cel putin un virf nu \
                este pe ecran\n");
        break;   // iesire din switch
        }
        if(cr=='D') {  // dreptunghi
            dr=new dreptunghi(a,c,YELLOW);
            if(dr -> validdrept()==0){
                printf(text1);
                printf(" dreptunghi ");
                printf(text2);
                break;   // iesire din switch
            }
            // se pune dreptunghiul in container daca este loc

            if(patrulater.atrib(dr)==0)
                i=EOF;   // container plin
            break;   // iesire din switch
        }
        else{  // patrat
            t=new patrat(a,c,YELLOW);
            if(t -> validpatrat()==0){
```

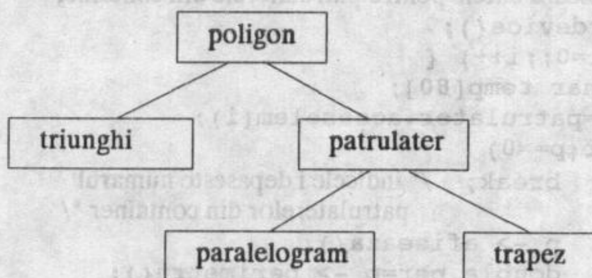
```
                    printf(text1);
                    printf(" patrat ");
                    printf(text2);
                    break; //iesire switch
                }
                // se pune patratul in container daca este loc
                if(patrulater.atrib(t)==0)
                    i=EOF;    //container plin
                break;   //iesire switch
            }   //sfirsit else
        } //sfirsit switch
        if(i==EOF)
            break;  //s-a intilnit sfirsitul de fisier
                    //sau containerul este plin
    }   //sfirsit for pentru citire date

    //se afiseaza datele pentru patrulaterele din container
    cleardevice();
    for(i=0;;i++) {
        char temp[80];
        p=patrulater.acceselem(i);
        if(p==0)
            break;    /* indicele i depaseste numarul
                         patrulaterelor din container */
            p -> afiseaza();
            double per=p -> perimetru();
            sprintf(temp,"perimetrul=%g",per);
            outtextxy(0,getmaxy()-20,temp);
            double ar=p -> arie();
            sprintf(temp,"aria=%g",ar);
            outtextxy(0,getmaxy()-10,temp);
            getch();
            cleardevice();   //sterg tot ecranul
    }
    closegraph();
  }
```

30.1. Clase abstracte

Clasele au fost introduse în limbajul C++ pentru a implementa tipuri noi de date, diferite de cele predefinite. Aceste tipuri corespund unor concepte care intervin în mod natural în programarea diferitelor aplicaţii

concrete. Aceste concepte au un caracter mai mult sau mai puţin general. O serie de concepte care intervin în rezolvarea unei probleme, nu sînt independente, prezentînd elemente comune. Se ajunge în felul acesta la un proces de ierarhizare a conceptelor, în vîrful ierarhiei aflîndu-se conceptul cel mai general, ale cărui proprietăţi sînt moştenite de celelalte concepte ale ierarhiei, mai puţin generale sau cum se obişnuieşte să se spună, mai specializate. O astfel de ierarhie, între conceptele utilizate în rezolvarea unei probleme, conduce la o ierarhie a claselor care implementează conceptele respective. Ierarhia dintre clase se defineşte cu ajutorul noţiunii de derivare. Dacă o clasă A este pe un nivel imediat inferior faţă de clasa B în ierarhia de clase, atunci se spune că clasa A este o clasă derivată a clasei B, iar despre B se spune că este o clasă de bază a lui A. Esenţa derivării constă în aceea că, elementele membru (date şi funcţii) ale clasei de bază sînt moştenite de clasa derivată. Clasele care corespund nivelelor mai înalte au un grad înalt de generalitate şi de aceea anumite funcţii membru ale acestora nu pot fi definite. De exemplu, să considerăm o ierarhie pentru figurile geometrice:



Un poligon are elemente comune pentru toate celelalte concepte ale ierarhiei şi anume:

- perimetru;
- arie;
- poate fi afişat;
- poate fi făcut invizibil după o afişare.

Aceste noţiuni relativ la *poligon* sînt prea generale şi ele sau nu pot fi definite cu ajutorul unor funcţii sau acestea chiar dacă se pot defini sînt ineficiente.

Ele se pot exprima în mod eficient numai la nivele mai inferioare ale ierarhiei.

De exemplu, pentru conceptul *triunghi* se pot defini funcţii pentru

elementele indicate mai sus. În mod analog, se pot defini funcţii pentru aceleaşi elemente şi în cazul conceptului *patrulater*. Unele pot fi definite mai eficient, dacă se coboară mai jos în ierarhie. Dacă un patrulater este un paralelogram sau un trapez, atunci calculul ariei se poate realiza mai eficient dacă se face separat pentru fiecare din aceste cazuri particulare.

Faptul că noţiunile de arie şi perimetru relativ la poligon sînt prea generale şi nu le definim decît pentru concepte mai specializate cum sînt triunghiul, paralelogramul, trapezul etc. ne conduce la noţiunea de *funcţie virtuală pură*. O *funcţie virtuală pură* este o funcţie virtuală care nu are definiţie ci numai prototip.

Funcţiile membru ale clasei *poligon* pentru calculul perimetrului sau ariei, precum şi cele pentru afişarea sau ştergerea unui poligon, sînt funcţii virtuale pure. Ele nu sînt definite pentru clasa poligon, fiind prea generale şi au numai prototipuri.

Funcţiile virtuale pure se definesc ca funcţii virtuale la nivele mai inferioare. În cazul de faţă, se pot defini funcţii virtuale corespunzătoare pentru clasele *triunghi* şi *patrulater* sau chiar pentru nivele mai particulare cum este *paralelogramul* sau *trapezul*.

Prototipul unei funcţii virtuale pure are unul din următoarele formate:

virtual *tip nume_funcţie* (*lista parametrilor formali*)=0;

sau

virtual *tip nume_funcţie* (*lista parametrilor formali*)**const**=0;

Cu alte cuvinte, o funcţie virtuală pură are un prototip care începe cu cuvîntul *virtual* şi se termină cu specificaţia:

=0

urmată de punctul şi virgula care termină orice prototip.

Prototipul unei funcţii virtuale pure se scrie în definiţia clasei pentru care este funcţie membru.

Presupunînd că clasa *poligon* are funcţiile virtuale pure amintite mai sus, vom defini această clasă astfel:

```
class poligon {
 public:
     virtual double perimetru() const = 0;
     virtual double arie() const = 0;
     virtual void afiseaza() = 0;
     virtual void sterge() = 0;
};
```

O clasă care are cel puţin o funcţie membru care este funcţie virtuală pură, se numeşte *clasă abstractă*.

Clasa *poligon*, definită ca mai sus, este un exemplu de clasă abstractă. Clasele abstracte nu au obiecte. Ele nu se pot utiliza la instanţieri ci numai ca şi clase de bază în procesul de derivare.

O funcţie virtuală pură a unei clase trebuie să fie suprapusă în orice clasă care derivă direct din clasa respectivă. Ea poate fi suprapusă ca o funcţie virtuală obişnuită sau tot ca o funcţie virtuală pură. În acest ultim caz, clasa derivată este şi ea o clasă abstractă.

Din cele de mai sus rezultă imediat rolul jucat de clasele abstracte şi de funcţiile virtuale pure.

O funcţie virtuală pură indică o funcţie a unei clase care *trebuie* să fie prezentă la orice clasă derivată din ea. Funcţia virtuală pură nu se defineşte deoarece clasa la care aparţine este prea generală pentru a putea fi definită sau se defineşte neeficient.

Clasele abstracte nu au instanţieri, deoarece ele au cel puţin o funcţie nedefinită. Acestea prezintă interes în procesul de derivare impunînd suprapunerea funcţiilor virtuale pure în clasele care derivă din ele.

Clasa *triunghi*, din ierarhia de mai sus, se defineşte astfel:

```
class triunghi:public poligon {
     punct tri[3];
 public:
     triunghi();
     triunghi(const punct&,const punct&,
              const punct&);
     triunghi(const triunghi&);
     double perimetru() const;
     double arie() const;
     void afiseaza();
     void sterge();
     triunghi& operator = (const triunghi&);
};
```

Funcţiile membru *perimetru*, *arie*, *afiseaza* şi *sterge* ale clasei *triunghi* sînt funcţii virtuale obişnuite care se suprapun peste funcţiile virtuale pure corespunzătoare clasei *poligon*. Aceste funcţii trebuie să fie definite în mod obişnuit.

Clasa *triunghi* nu mai este o clasă abstractă, deci ea poate fi instanţiată:

```
triunghi abc;
```

```
punct p(1,2),q(3,2),r(2,3);
triunghi pqr(p,q,r);
```

sînt instanţieri corecte ale clasei *triunghi*.

O instanţiere de forma:

```
poligon pol;
```

este eronată, clasa *poligon* fiind o clasă abstractă.

Clasa *patrulater* poate fi definită tot ca o clasă abstractă, ca mai jos:

```
class patrulater:public poligon {
 protected:
     punct patru[4];
 public:
     patrulater();
     patrulater(const punct& a,const punct& b,
                const punct& c,const punct& d);
     patrulater(const patrulater&);
     double perimetru() const = 0;
     double arie() const = 0;
     void afiseaza();
     void sterge();
     patrulater& operator = (const patrulater&);
};
```

Clasa *patrulater* are două funcţii virtuale pure: *perimetru* şi *arie*. Acestea urmează a fi suprapuse prin funcţii virtuale obişnuite (care nu sînt pure) atît în clasa *paralelogram* cît şi în clasa *trapez*.

Funcţiile *afiseaza* şi *sterge*, ale clasei *patrulater*, sînt funcţii virtuale obişnuite care se definesc folosind funcţia standard *drawpoly* care afişează un poligon (vezi paragraful 19.7.).

Un alt exemplu de utilizare a claselor abstracte este cel al ierarhiei care implementează conceptul de *grupoid*.

O mulţime oarecare de elemente formează un *grupoid* dacă pe mulţimea respectivă este definită o lege de compoziţie internă. De aici, rezultă că pentru a defini un grupoid este necesar să precizăm mulţimea şi legea de compoziţie internă.

Spre exemplificare, vom considera mulţimea numerelor întregi, iar ca lege internă alegem operaţia de scădere. Contrar practicii încetăţenite în matematică, alegînd caracterul "+" pentru a nota legea de compoziţie internă, a+b va însemna nu suma dintre *a* şi *b*, ci chiar diferenţa lor.

Primul pas în definirea grupoidului va fi implementarea unei submulţimi a mulţimii numerelor întregi. Aceasta se face printr-o clasă pe care o denumim *elem*. Fixîndu-ne asupra mulţimii numerelor întregi de tip *long*, înseamnă că clasa *elem* va avea o dată membru de tip *long*, pe care o numim *gelem*.

Ca funcţii membru vom avea şi o funcţie care permite acces la data membru *gelem*.

Definim clasa *elem* ca mai jos:

```
class elem {
 protected:
    long gelem;
 public:
    elem()    //constructor implicit
    {
    }

    elem(long g)
    //constructor utilizat la initializari si conversii
    {
       gelem = g;
    }

    elem(const elem& e)
    //constructor de copiere
    {
       gelem = e.gelem;
    }

    long retgelem()
    {
       return gelem;
    }
 };
```

Pasul următor, în definirea grupoidului, este de a defini o lege de compoziţie internă pe mulţimea obiectelor clasei *elem*, lege pe care convenim să o notăm cu "+".

În matematică afirmăm despre o mulţime că este grupoid dacă ştim că pe mulţimea respectivă este definită o lege de compoziţie internă, făcînd abstracţie de modul în care este definită legea respectivă. Acelaşi lucru îl putem exprima în cazul de faţă definind clasa *grupoid* ca o clasă abstractă, care derivă din clasa *elem*. Această clasă, în afară de constructori, are o

funcţie virtuală pură care supraîncarcă operatorul "+" pentru obiecte de tip *elem*. Aceasta înseamnă că orice clasă derivată din clasa *grupoid* trebuie să supraîncarce operatorul "+". La acest nivel se pot defini clase care să nu mai fie abstracte, avînd definit modul concret de supraîncărcare al operatorului "+".

Clasa *grupoid* poate fi definită astfel:

```
class grupoid:public elem {   //clasa abstracta
 public:
     grupoid()   //constructor implicit
     {
         gelem = 0;
     }

     grupoid(long g):elem(g)
    //constructor utilizat la conversii
     {
     }

     grupoid(elem e):elem(e)
    //constructor utilizat la conversii
     {
     }

     virtual elem operator + (elem) = 0;
    /* - functie virtuala pura;
       - supraincarca operatorul +pentru obiecte de tip elem;
       orice clasa derivata din clasa  grupoid trebuie sa supraincarce
       operatorul +. */
};
```

În momentul de faţă putem implementa grupoidul anunţat mai sus, care este format din mulţimea numerelor de tip întreg, legea de compoziţie fiind operaţia de scădere. Notăm cu *Zminus* clasa derivată din clasa *grupoid* şi ale cărei obiecte formează grupoidul dorit.

Clasa *Zminus* are o singură dată membru şi anume pe *gelem* care este moştenită de la clasa *elem* şi este de tip *long*.

Funcţiile membru ale clasei *Zminus* sînt constructorii, funcţia care supraîncarcă operatorul "+" şi eventual alte funcţii care intervin în utilizarea obiectelor, cum ar fi, de exemplu, funcţia de afişare a datei membru *gelem* moştenită de la clasa *elem*.

Definim clasa *Zminus* astfel:

```
class Zminus:public grupoid {
  public:
      Zminus()   // constructor implicit
      {
      }

      Zminus(long zm):grupoid(zm)
      // constructor utilizat la initializare si conversie
      {
      }

      Zminus(elem e):grupoid(e)
      // constructor utilizat la initializare si conversie
      {
      }

      elem operator + (elem zm)
      // supraincarca operatorul +
      {
          elem v(gelem-zm.retgelem());
          return v;
      }

      void afis()
      // afiseaza data membru gelem a obiectului curent
      {
          printf("%ld\n",gelem);
      }
};
```

Fie instanţierile:

```
Zminus z1(123456789);
Zminus z2(123000000);
```

O expresie de forma:

```
z1+z2
```

este corectă şi ea are ca valoare un obiect de tip *elem* pentru care *gelem* are valoarea:

z1.gelem-z2.gelem = 123456789-123000000 = 456789

Pentru a verifica acest lucru, putem folosi secvenţa de mai jos:

```
Zminus x = z1+z2;
```

```
x.afis();
```

Pentru a aplica operatorul "+", supraîncărcat ca mai sus, la obiecte ale clasei *elem*, putem aplica explicit constructorul clasei *Zminus* care converteşte un obiect de tip *elem* într-un obiect de tip *Zminus*.

Fie, de exemplu, instanţierile:

```
elem a(123456789);
elem b(123000000);
```

O expresie de forma:

```
a+b
```

este eronată, deoarece obiectul din stînga operatorului trebuie să fie o instanţiere a clasei *Zminus*. De aceea, expresia de mai sus, se scrie apelînd explicit conversia din tipul *elem* în tipul *Zminus*:

```
Zminus(a)+b
```

Mulţimea obiectelor clasei *elem* reprezintă o submulţime a numerelor întregi şi anume ea corespunde numerelor întregi din intervalul [-2**31,2**31-1].

Obiectele acestei clase pot fi utilizate în expresii de forma:

(1) Zminus(a)+b

unde:

a şi b - Sînt instanţieri ale clasei *elem*.

Expresia (1) are ca rezultat un obiect al clasei *elem*, a cărei dată membru are ca valoare diferenţa dintre datele membru ale obiectelor *a* şi *b*. În felul acesta, operatorul "+" din expresia (1), corespunde operaţiei de scădere definită pe mulţimea numerelor întregi. De aceea, se poate considera că mulţimea obiectelor clasei *elem* formează un grupoid faţă de operaţia "+" definită prin funcţia membru *operator* + a clasei *Zminus*.

Acest *grupoid* corespunde *grupoidului* format de mulţimea numerelor întregi faţă de operaţia de scădere definită pe mulţimea respectivă.

Observaţie:

Grupoidul format de obiectele clasei *elem* nu corespunde în mod riguros cu noţiunea de *grupoid* definită în matematică pe mulţimea numerelor întregi faţă de operaţia de scădere, deoarece expresia (1) are o valoare eronată cînd rezultatul aplicării operatorului "+" ne conduce la o valoare care este în afara limitelor tipului *long*.

Schimbînd definiţia supraîncărcării operatorului "+", se pot implementa

alte grupoide pe mulţimea obiectelor clasei *elem*.

Fie de exemplu, clasa *Zplus*, care diferă de clasa *Zminus* numai prin funcţia *operator* +:

```
class Zplus:public grupoid {
 public:
    ...
    elem operator + (elem zp)
    // supraincarca operatorul +
    {
        elem v(gelem + zp.retgelem());
        return v;
    }
    ...
};
```

În acest caz, obiectele clasei *elem* formează un grupoid care corespunde grupoidului definit pe mulţimea numerelor întregi faţă de adunare. Acest grupoid formează chiar un grup, spre deosebire de cel precedent, care nu formează nici măcar un semigrup, deoarece operaţia de scădere nu este asociativă.

Obiectele clasei *Zminus* pot şi ele forma un grupoid faţă de operaţia "+", dacă se adaugă, la clasa respectivă, funcţia membru de mai jos:

```
Zminus Zminus::operator + (Zminus zm)
// supraincarca operatorul + pentru obiectele clasei Zminus
{
 Zminus v(*this+elem(zm));
 return v;
}
```

Fie instanţierile:

```
Zminus a(123456789);
Zminus b(123000000);
```

Expresia:

```
a+b
```

este corectă şi ea se evaluează apelîndu-se funcţia *operator*+ de mai sus, supraîncărcată pentru obiectele clasei *Zminus*.

În felul acesta, se poate afirma că obiectele clasei *Zminus* formează şi ele un *grupoid* care corespunde cu grupoidul format de elementele clasei *elem*.

Declaraţia:

```
Zminus v(*this+elem(zm));
```

din corpul funcţiei care supraîncarcă operatorul + conţine termenul:

```
elem(zm)
```

Acest termen apelează constructorul de copiere al clasei *elem* deoarece *zm* este considerat un obiect "particular" al clasei *elem*, fiind o instanţiere a clasei *Zminus* care derivă indirect din clasa *elem*.

La evaluarea expresiei:

```
*this+elem(zm)
```

se apelează funcţia care supraîncarcă operatorul +, de prototip:

```
elem operator + (elem);
```

Să observăm că, expresia:

```
*this+zm
```

este corectă din punct de vedere sintactic, dar utilizarea ei conduce la un apel recursiv infinit al funcţiei de prototip:

```
Zminus operator + (Zminus);
```

În încheierea acestui paragraf amintim restricţiile legate de utilizarea claselor abstracte.

Aşa cum s-a amintit mai sus, o clasă abstractă poate fi utilizată numai ca o clasă de bază. Nu se pot face instanţieri pentru o clasă abstractă.

O clasă abstractă nu poate fi tipul unui parametru. O funcţie nu poate returna un obiect al unei clase abstracte.

Un parametru poate fi o referinţă la o clasă abstractă. O funcţie poate returna o referinţă la o clasă abstractă. Referinţele la o clasă abstractă sînt admise numai dacă la utilizarea lor nu se crează obiecte intermediare.

În sfîrşit, să amintim că se pot declara pointeri de tipuri implementate prin clase abstracte.

Exerciţii:

30.7 Să se implementeze conceptul *poligon* printr-o clasă abstractă.

FIŞIERUL BXXX7.H

```
class poligon {
 public:
    virtual int polvalid() = 0;
    virtual double perimetru() const = 0;
```

```
      virtual double arie() const = 0;
      virtual void afiseaza() = 0;
      virtual void sterge() = 0;
  };
```

30.8 Să se implementeze conceptul de *triunghi* printr-o clasă derivată din clasa abstractă *poligon* definită în exercițiul 30.7.

FIŞIERUL BXXX8.H

```
  #ifndef __GRAPHICS_H
  #include <graphics.h>
  #define __GRAPHICS_H
  #endif
  #ifndef __GSEGMENT_H
  #include "BXXIX11.CPP"      // clasa gsegment
  #define __GSEGMENT_H
  #endif
  #ifndef __POLIGON_H
  #include "BXXX7.H"          // clasa poligon
  #define __POLIGON_H
  #endif

  class triunghi:public poligon {
   protected:
      punct tri[3];
      int tvizibil,tvalid;
   public:
      int polvalid();
      /* - returneaza 1 daca tri[0],tri[1],tri[2] nu sint coliniare;
         - returneaza 0 altfel.
      */

      triunghi();   // constructor implicit
      triunghi(double xa,double ya,double xb,
               double yb,double xc,double yc);
      // constructor utilizat la initializari

      triunghi(const punct& a,const punct& b,
               const punct& c);
      // constructor utilizat la initializari

      triunghi(const triunghi&);  // constructor de copiere
```

```
    double perimetru()const;
    // returneaza perimetrul triunghiului curent

    double arie()const;
    // returneaza aria triunghiului curent

    void afiseaza();
    // afiseaza triunghiul curent

    void sterge();
    // face invizibil triunghiul curent
};
```

Funcţiile membru se definesc în fişierul de mai jos, de extensie *CPP*.

FIŞIERUL BXXX8

```
#ifndef __TRIUNGHI_H
#include "BXXX8.H"
#define __TRIUNGHI_H
#endif
#ifndef __MATH_H
#include <math.h>
#define __MATH_H
#endif

int triunghi::polvalid()
/* - returneaza 1 daca virfurile obiectului curent formeaza un
     triunghi (nu sint coliniare);
   - returneaza 0 altfel.
*/
{
 gsegment ab(tri[0],tri[1]);

 if((ab & tri[2]) == 0) {
     tvalid = 1;
     return 1;   // virfurile formeaza un triunghi
 }
 tvalid = 0;
 return 0;   // virfurile sint coliniare
}

inline triunghi::triunghi()  // constructor implicit
{
 tvalid = 0;
```

```
   tvizibil = 0;
}

triunghi::triunghi(double xa,double ya,
         double xb,double yb,double xc,double yc)
// constructor utilizat la initializari
{
 tri[0] = punct(xa,ya);
 tri[1] = punct(xb,yb);
 tri[2] = punct(xc,yc);
 polvalid();
 tvizibil = 0;
}

triunghi::triunghi(const punct& a,
              const punct& b,const punct& c)
// constructor utilizat la initializari
{
 tri[0] = a;
 tri[1] = b;
 tri[2] = c;
 polvalid();
 tvizibil = 0;
}

triunghi::triunghi(const triunghi& t)
// constructor de copiere
{
 for(int i=0;i < 3;i++)
     tri[i]=t.tri[i];
 polvalid();
 tvizibil=t.tvizibil;
}

double triunghi::perimetru() const
// returneaza perimetrul triunghiului curent
{
 if(tvalid){
     double p=tri[0].dist(tri[1]);
     p = p+tri[1].dist(tri[2]);
     p = p+tri[2].dist(tri[0]);
     return p;
 }
```

```
 return 0;    //virfurile nu formeaza un triunghi
}

double triunghi::arie() const
//returneaza aria triunghiului curent
{
 if(tvalid) {      //se aplica formula lui Heron
    double a = tri[1].dist(tri[2]);
    double b = tri[0].dist(tri[2]);
    double c = tri[0].dist(tri[1]);
    double p = (a+b+c)/2;   //semiperimetrul
    return sqrt(p*(p-a)*(p-b)*(p-c));
 }
 return 0;    //virfurile nu formeaza un triunghi
}

void triunghi::afiseaza()
//afiseaza triunghiul curent
{
 int tri1[8];   //tabel cu coordonate
 int i,j;

 if(tvalid){
    for(i=0,j=0; i < 6;i += 2,j++){
        tri1[i] = tri[j].retabs();
        tri1[i+1] = tri[j].retord();
    }
    tri1[6] = tri[0].retabs();
    tri1[7] = tri[0].retord();
    drawpoly(4,tri1);
    tvizibil=1;
 }
}

void triunghi::sterge()
//face invizibil triunghiul curent
{
  int c = getcolor();

  if(tvalid){
    setcolor(getbkcolor());
    afiseaza();
    setcolor(c);
```

```
        tvizibil = 0;
    }
}
```

30.9 Să se implementeze conceptul de *patrulater* printr-o clasă derivată din clasa abstractă *poligon* care are două funcţii virtuale pure: una pentru calculul perimetrului şi cealaltă pentru calculul ariei. Funcţiile membru *afiseaza* şi *sterge* se definesc la fel ca în cazul clasei *triunghi*, definită în exerciţiul 30.8.

FIŞIERUL BXXX9.H

```
#ifndef __GRAPHICS_H
#include <graphics>
#define __GRAPHICS_H
#endif
#ifndef __GSEGMENT_H
#include "BXXIX11.CPP"      // clasa gsegment
#define __GSEGMENT_H
#endif
#ifndef __POLIGON_H
#include "BXXX7.H"          // clasa poligon
#define __POLIGON_H
#endif

class patrulater:public poligon {
 protected:
     punct patru[4];
     int valid;
     int pvizibil;
 public:
     int polvalid()=0;
     patrulater();   // constructor implicit
     patrulater(double xa,double ya,double xb,
                double yb,double xc,double yc,
                double xd,double yd);
     // constructor utilizat la initializari

     patrulater(const punct& a,const punct& b,
                 const punct& c,const punct& d);
     // constructor utilizat la initializari

     patrulater(const patrulater&);
     // constructor de copiere
```

```
    virtual double perimetru() const = 0;
    // functie virtuala pura

    virtual double arie() const = 0;
    // functie virtuala pura

    void afiseaza();

    void sterge();
};
```

Funcţiile membru se definesc în fişierul de mai jos, de extensie *CPP*.

FIŞIERUL BXXX9

```
#ifndef __PATRULATER
#include "BXXX9.H"      // clasa patrulater
#define __PATRULATER
#endif

inline patrulater::patrulater()  // constructor implicit
{
 pvizibil = 0;
 valid = 0;
}

patrulater::patrulater(double xa,double ya,
               double xb,double yb, double xc,
               double yc,double xd,double yd)
// constructor utilizat la initializari
{
 patru[0] = punct(xa,ya);
 patru[1] = punct(xb,yb);
 patru[2] = punct(xc,yc);
 patru[3] = punct(xd,yd);
 valid = 1;
 pvizibil = 0;
}

patrulater::patrulater(const punct& a,
      const punct& b,const punct& c,const punct& d)
// constructor utilizat la initializari
{
 patru[0] = a;
 patru[1] = b;
```

```
 patru[2] = c;
 patru[3] = d;
 valid = 1;
 pvizibil = 0;
}

patrulater::patrulater(const patrulater& p)
// constructor de copiere
{
 for(int i=0;i < 4;i++)
     patru[i]=p.patru[i];
 valid = p.valid;
 pvizibil = p.pvizibil;
}

void patrulater::afiseaza()
// afiseaza patrulaterul curent
{
 int patrul[10];
 int i,j;

 if(valid){
     for(i=0,j=0; i < 8;i += 2,j++){
         patrul[i] = patru[j].retabs();
         patrul[i+1] = patru[j].retord();
     }
     patrul[8] = patru[0].retabs();
     patrul[9] = patru[0].retord();
     drawpoly(5,patrul);
     pvizibil=1;
 }
}

void patrulater::sterge()
// face invizibil patrulaterul curent
{
 int c = getcolor();

 if(valid){
     setcolor(getbkcolor());
     afiseaza();
     setcolor(c);
     pvizibil = 0;
```

```
    }
  }
```

30.10 Să se implementeze conceptul de *paralelogram* printr-o clasă derivată din clasa *patrulater* definită în exercițiul 30.9.

Obiectele clasei paralelogram pot fi afişate sau şterse folosind funcţiile membru *afiseaza* şi respectiv *sterge* ale clasei *patrulater*.

FIŞIERUL BXXX10.H

```
#ifndef __PATRULATER_H
#include "BXXX9.CPP"    // clasa patrulater
#define __PATRULATER_H
#endif
#ifndef __MATH_H
#include <math.h>
#define __MATH_H
#endif

class paralelogram:public patrulater {
 protected:
     int pvalid;
 public:
     int polvalid();
     /* - returneaza 1 daca obiectul curent este un paralelogram;
        - returneaza 0 altfel.
     */

     paralelogram();   // constructor implicit
     paralelogram(double xa,double ya,double xb,
                  double yb,double xc,double yc,
                  double xd,double yd);
     // constructor utilizat la initializari

     paralelogram(const punct& a,const punct& b,
                  const punct& c,const punct& d);
     // constructor utilizat la initializari

     paralelogram(const paralelogram&);
     // constructor de copiere

     double perimetru() const;
     // returneaza perimetrul paralelogramului curent
```

```
    double arie() const;
    // returneaza aria paralelogramului curent
};
```

Mai jos, se definesc funcţiile membru într-un fişier de extensie *CPP*.

FIŞIERUL BXXX10

```
#ifndef __PARALELOGRAM
#include "BXXX10.H"    // clasa paralelogram
#define __PARALELOGRAM
#endif

int paralelogram::polvalid()
/* - returneaza 1 daca obiectul curent este un paralelogram;
   - returneaza 0 altfel.
*/
{
 gsegment ab(patru[0],patru[1]);
 gsegment bc(patru[1],patru[2]);
 gsegment cd(patru[2],patru[3]);
 gsegment da(patru[3],patru[0]);

 if((ab == bc) || (ab == cd) || (ab == da) ||
    (bc == cd) || (bc == da) || (cd == da)) {
    pvalid = 0;
    return 0;
 }
 if(ab || cd)
    if(bc||da){
       pvalid = 1;
       return 1;
    }
 pvalid = 0;
 return 0;
}

inline paralelogram::paralelogram() // constructor implicit
{
 pvalid = 0;
}
```

```
inline paralelogram::paralelogram(double xa,
        double ya,double xb,double yb,doublxe,d
         double yc,double xd,double yd):
     patrulater(xa,ya,xb,yb,xc,yc,xd,yd)
// constructor utilizat la initializari
{
 polvalid();
}

inline paralelogram::paralelogram(const punct& a,
             const punct& b, const punct& c,
              const punct& d):
      patrulater(a,b,c,d)
// constructor utilizat la initializari
{
 polvalid();
}

paralelogram::paralelogram(const paralelogram& p)
// constructor de copiere
{
 for(int i=0;i < 4;i++)
    patru[i]=p.patru[i];
 polvalid();
 pvizibil = p.pvizibil;
}

double paralelogram::perimetru() const
// returneaza perimetrul paralelogramului curent
{
 if(pvalid){
    double ab = patru[0].dist(patru[1]);
    double bc = patru[1].dist(patru[2]);
    return 2*(ab+bc);
 }
 return 0;
}

double paralelogram::arie() const
// returneaza aria paralelogramului curent
{
 if(pvalid) {
    gsegment CD(patru[2],patru[3]);
```

```
   gpunct A(patru[0]);
   //-se determina inaltimea paralelogramului din punctul A
   gsegment I = CD | A;
   punct E=I*CD;  // intersectia inaltimii din A cu latura CD
   double cd=patru[2].dist(patru[3]);
   // lungimea laturii CD

   double ae = E.dist(A);  // lungimea inaltimii AE
   return cd*ae;
 }
 return 0;
}
```

30.11 Să se scrie un program care afişează un triunghi şi un paralelogram pe ecranul grafic împreună cu perimetrul şi aria figurilor respective. Afişările se fac secvenţial, întîi triunghiul şi apoi paralelogramul, după acţionarea unei taste ASCII oarecare.

PROGRAMUL BXXX11

```
#include <conio.h>
#include "BXXX8.CPP"     // clasa triunghi
#include "BXXX10.CPP"   // clasa paralelogram

main()
/* - afiseaza un triunghi si un paralelogram impreuna cu perimetrele
     si ariile lor;
   - afisarea se face secvential, intii triunghiul si apoi paralelogramul,
     dupa actionarea unei taste ASCII oarecare.
*/
{
 triunghi t(80,10,100,210,50,150);
 paralelogram p(100,100,200,200,240,220,140,120);
 char tab[80];
 int gd = DETECT,gm;
 initgraph(&gd,&gm,"c:\\BORLANDC\\BGI");
 t.afiseaza();
 double per = t.perimetru();
 sprintf(tab,"perimetrul triunghiului = %g",per);
 outtextxy(0,320,tab);
 double ar=t.arie();
 sprintf(tab,"aria triunghiului = %g",ar);
 outtextxy(0,330,tab);
```

```
getch();
cleardevice();
p.afiseaza();
per = p.perimetru();
sprintf(tab,"perimetrul paralelogramului =\
       %g",per);
outtextxy(0,320,tab);
ar=p.arie();
sprintf(tab,"aria paralelogramului=%g",ar);
outtextxy(0,330,tab);
getch();
closegraph();
}
```

31. INTRĂRI/IEŞIRI

Limbajul C++, la fel ca limbajul C, nu are instrucţiuni specifice operaţiilor de intrare/ieşire.

În limbajul C, astfel de operaţii se realizează cu ajutorul unui set de funcţii din biblioteca standard a sistemului. Aceste funcţii pot fi folosite în acelaşi mod şi în programele scrise în C++.

În afara acestor funcţii, biblioteca standard a limbajului C++ oferă utilizatorului ierarhii de clase care permit realizarea de astfel de operaţii.

La baza acestor operaţii se află conceptul de *stream* (flux). Prin *stream* se înţelege un flux de date de la o *sursă* la o *destinaţie* sau *consumator*.

În general, sursa de date poate fi tastatura, un fişier de pe disc sau chiar o zonă de memorie. În mod analog, destinaţia poate fi ecranul, un fişier pe disc sau o zonă de memorie.

Biblioteca standard a limbajului C++ conţine două ierarhii de clase, una are ca rădăcină clasa *streambuf*, iar cealaltă, clasa *ios*.

Clasa *streambuf* furnizează funcţii generale pentru lucrul cu zonele tampon (bufere) şi permite tratarea operaţiilor de intrare/ieşire fără a avea în vedere formatări complexe. Din clasa *streambuf* derivă clasele *strstreambuf* şi *filebuf*.

Clasa *ios* are un pointer spre *streambuf*. Ea are date membru pentru a gestiona interfaţa cu *streambuf* şi pentru tratarea erorilor. Din clasa *ios* derivă două clase: *istream* pentru gestiunea intrărilor şi *ostream* pentru gestiunea ieşirilor. Aceste derivări se realizează utilizînd clasa *ios* drept clasă de bază virtuală:

class istream:virtual public ios

şi

class ostream:virtual public ios

O a treia clasă, numită *iostream* derivă din ambele clase:

class iostream:public istream,public ostream

Clasele *istream, ostream* şi *iostream* sînt, fiecare, clase de bază pentru alte trei derivări şi anume:

class istream_withassign:public istream
class ostream_withassign:public ostream
class iostream_withassign:public iostream

Clasa *istream* permite realizarea de conversii formatate sau neformatate ale caracterelor încărcate din obiecte de tip *streambuf*.

Clasa *ostream* permite conversii formatate sau neformatate în caractere care se păstrează în obiecte de tip *streambuf*.

Clasa *iostream* moşteneşte facilităţile ambelor clase de bază permiţînd operaţii în ambele direcţii.

Clasele cu sufixul *withassign* furnizează "stream"-uri (fluxuri de date) standard: *cin, cout, cerr* şi *clog*.

Obiectul *cin* (console input) este o instanţiere a clasei *istream_withassign* şi el corespunde fişierului standard de intrare definit de pointerul *stdin*.

În mod analog, obiectul *cout* (console output) este o instanţiere a clasei *ostream_withassign* şi el corespunde fişierului standard de ieşire definit de pointerul *stdout*.

Obiectele *cerr* şi *clog* sînt şi ele instanţieri ale clasei *ostream_withassign* şi corespund fişierului standard de ieşire definit de pointerul *stderr*. Acestea au facilităţi suplimentare faţă de fişierul *stderr*.

Clasele cu sufixul *withassign* se deosebesc de clasele lor de bază prin aceea că ele supraîncarcă operatorul de atribuire.

De exemplu, clasa *istream_withassign* se defineşte astfel:

```
class istream_withassign:public istream {
 istream_withassign();
 istream& operator = (istream&);
 istream& operator = (streambuf&);
};
```

O *clasă stream* este o clasă derivată din *istream* sau *ostream*.

Utilizarea ierarhiilor de clase amintite mai sus implică includerea fişierului *iostream.h*. Aşa cum se va vedea mai tîrziu, în cazuri mai complexe de utilizare a acestor ierarhii este nevoie să se includă în program şi alte fişiere de extensie *h*.

Utilizarea facilităţilor oferite de clase, conduce la o flexibilitate şi claritate mai mare în realizarea operaţiilor de intrare/ieşire faţă de funcţiile standard obişnuite: *scanf, printf, gets, puts* etc.

Un avantaj imediat al utilizării *stream*-urilor este acela de a elimina erorile care apar în mod frecvent la apelul funcţiei *printf*, cînd numărul specificatorilor de format diferă de cel al parametrilor efectivi. La aceasta adăugăm şi posibilităţile acestor clase de a realiza operaţii de intrare/ieşire nu numai cu date de tipuri predefinite, ci şi cu obiecte de tipuri abstracte.

Utilizatorul are posibilitatea să definească simplu conversii pentru obiectele asupra cărora se realizează operaţii de intrare/ieşire.

Obiectele care sînt instanţieri ale claselor ierarhiei care are ca rădăcină clasa *ios* se numesc *stream*-uri. În particular, obiectele *cin, cout, cerr* şi *clog* sînt *stream*-uri standard.

31.1. Intrări/Ieşiri standard

Prin intrări/ieşiri standard înţelegem schimbul de informaţii care se realizează cu terminalul de la care s-a lansat programul. Aşa cum s-a afirmat mai sus, se pot folosi 4 obiecte standard pentru a realiza intrări/ieşiri standard: *cin, cout, cerr* şi *clog*.

Obiectul *cin* este o instanţiere a clasei *istream_withassign*, iar celelalte sînt instanţieri ale clasei *ostream_withassign*. Aceste obiecte sînt iniţializate prin includerea fişierului *iostream.h*. La includerea acestui fişier se apelează constructorii claselor mai sus amintite.

Obiectele *cin, cout, cerr* şi *clog* se instanţiază şi iniţializează o singură dată, chiar dacă fişierul respectiv se include de mai multe ori. În felul acesta, la lansarea programului, obiectele respective sînt disponibile şi pot fi utilizate pentru a realiza operaţii de intrare/ieşire de la terminalul standard.

Menţionăm că aceste operaţii pot fi redirectate spre alte periferice, la fel cum se procedează cu funcţiile *scanf*, *printf* etc.

31.1.1. Ieşire standard

Ieşirile standard se pot realiza folosind operatorul "<<". Acesta este supraîncărcat pentru operaţii de ieşire. Operandul stîng este un obiect al clasei *ostream*.

Operandul din dreapta are un tip pentru care a fost supraîncărcat operatorul "<<". Acesta poate fi un tip predefinit sau un tip abstract.

Operatorul "<<" supraîncărcat pentru operaţii de ieşire se numeşte *operator de inserare* (insertion) sau *pune la ...* (put to).

Prototipul funcţiei membru a clasei *ostream* care supraîncarcă operatorul de inserare este următorul:

ostream& *operator* **<<** *(tip)*;

În mod implicit, operatorul de inserare este supraîncărcat pentru tipurile predefinite: *char* (cu semn sau fără semn), *short, int, long, unsigned, unsigned long, char* * (se utilizează pentru şiruri de caractere), *float,*

double, long double şi *void* * (pentru pointeri, ieşirea se face în hexazecimal).

Amintim că operatorul "<<" este în mod normal operatorul de deplasare la stînga. El are prioritatea imediat mai mică decît operatorii binari aditivi (+ şi -) şi imediat mai mare decît operatorii relaţionali. Operatorii de deplasare se asociază de la stînga la dreapta. Aceste reguli rămîn valabile şi pentru operatorul de inserare. Deoarece la supraîncărcarea operatorului de inserare se returnează o referinţă la obiectul curent, operatorii de inserare se pot aplica înlănţuiţi.

Exemple:

```
int i;
...
cout << i;   // afiseaza valoarea lui i

// are acelasi efect cu apelul
printf("%d",i);
...
cout << "\n";   // afiseaza newline

// are acelasi efect cu apelul
printf("\n");

// sau
putchar('\n');

double d;
...
cout << d;      // afiseaza valoarea lui d

// are acelasi efect cu apelul
printf("%g",d);

char c;
...
cout << c;      // afiseaza caracterul al carui cod ASCII
                // coincide cu valoarea lui c
// are acelasi efect cu apelul
printf("%c",c);

char t[] = "Acesta este un sir de caractere";
...
cout << t;   // afiseaza sirul spre care pointeaza t
```

```
// are acelasi efect cu apelul
printf("%s",t);

int a;
...
cout << &a;      // afiseaza, in hexazecimal, adresa zonei alocate lui a
cout << i+di;    // afiseaza valoarea expresiei i+di;
                 // operatorul << este mai putin prioritar decit +
cout << (i&0377); // afiseaza valoarea octetului mai putin
                  // semnificativ a lui i
```

În ultima instructiune sînt necesare parantezele rotunde datorită faptului că operatorul "<<" este mai prioritar decît "&".

Instrucţiunea expresie:

```
cout << "i=" << i;
```

afişează:

i = valoarea lui i

Are acelaşi efect cu apelul:

```
printf("i=%d",i);
```

Instrucţiunea:

```
cout << "i=" << i+10 << "\n";
```

are acelaşi efect cu apelul:

```
printf("i=%d\n",i+10);
```

Funcţia *printf* oferă facilităţi de formatare complexă a datelor de ieşire. Astfel de formatări sînt posibile şi în cazul operatorului de inserare. În acest caz, se utilizează diferiţi indicatori de format. Aceştia sînt definiţi în clasa *ios*, ca mai jos:

```
public:
 enum{
     skipws=0x0001,  // salt peste caractere albe la intrare
     left=0x0002,    // iesire cadrata in stinga
     right=0x0004,   // iesire cadrata in dreapta
     internal=0x0008, // caractere nesemnificative dupa semn
                      // sau indicator de baza
     dec=0x0010,     // conversie in zecimal
     oct=0x0020,     // conversie in octal
     hex=0x0040,     // conversie in hexazecimal
```

```
        showbase=0x0080,      // se afiseaza baza
        showpoint=0x0100,     // se afiseaza punctul zecimal
        uppercase=0x0200,     // iesire hexazecimala cu litere mari
        showpos=0x0400,       // intregii pozitivi sint afisati cu + in fata
        scientific=0x0800,    // numerele flotante se afiseaza cu
                              // exponent
        fixed=0x1000,         // numerele flotante se scriu fara exponent
        unitbuf=0x2000,       // se videaza zonele tampon pentru toate
                              // inserarile
        stdio=0x4000,         // dupa fiecare inserare se videaza stdout
                              // si stderr
    };
    ...
```

Aceşti enumeratori reprezintă biţi din dublul cuvînt *x_flag* care este dată membru a clasei *ios*. Aceşti biţi sînt, în mod implicit, aşa încît să se asigure formate standard la ieşire. Acestea sînt aceleaşi cu formatele definite de următorii specificatori de format utilizaţi la apelul funcţiei *printf* sau *scanf*:

- %d pentru **int**;
- %ld pentru **long**;
- %u pentru **unsigned**;
- %lu pentru **unsigned long**;
- %g şi %lg pentru **float** şi **double**;
- %Lg pentru **long double**;
- %c pentru **char**;
- %s pentru şiruri de caractere.

Utilizatorul poate seta biţii din data membru *x_flag* pentru a realiza formatări diferite decît cea implicită.

Setările biţilor din *x_flag* se fac cu funcţii membru ale claselor ierarhiei care au ca rădăcină clasa *ios*.

În afară de setările biţilor din data membru *x_flag*, la definirea formatelor, utilizatorul are posibilitatea de a defini dimensiunea cîmpului, la fel ca în cazul specificatorilor de format utilizaţi la apelul funcţiilor *printf* şi *scanf*.

În mod implicit, la afişarea unei date, cîmpul de afişare are atîtea caractere cîte sînt necesare pentru afişarea datei respective. Utilizatorul poate impune un cîmp minim pentru afişare cu ajutorul funcţiei membru

width a clasei *ios*.

Funcţia *width* are două prototipuri:

```
int width();
```

şi

```
int width(int w);
```

Apelul fără parametru al funcţiei *width* returnează dimensiunea curentă a cîmpului. Ea este definită de data membru *x_width* a clasei *ios*.

În mod implicit, valoarea dimensiunii cîmpului este zero, ceea ce înseamnă că data se afişează pe un cîmp de lungime egală cu numărul de caractere necesar afişării ei.

Apelul funcţiei *width* cu parametrul *w* permite utilizatorului să definească lungimea minimă a cîmpului în care se realizează afişarea. În acest caz, funcţia *width* returnează dimensiunea curentă a cîmpului înaintea modificării ei la valoarea lui *w*.

Dacă dimensiunea curentă a cîmpului este mai mică decît numărul de caractere al datei care se afişează, atunci data respectivă se va afişa fără trunchiere, adică pe atîtea caractere cîte sînt necesare. Dacă dimensiunea curentă a cîmpului este mai mare decît numărul de caractere al datei care se afişează, atunci data respectivă se afişează în cîmpul respectiv completată cu caractere nesemnificative pînă la dimensiunea cîmpului. În mod implicit data se afişează cadrată în dreapta în cîmpul curent, iar caracterele nesemnificative sînt spaţii. În felul acesta, funcţia *width* asigură compatibilitate pentru interpretarea lungimii cîmpului curent la operaţia de inserare cu cea definită de specificatorii de format utilizaţi la apelul funcţiei *printf*.

Exemplu:

```
int i;
...
i=123;
...
cout.width(5); //se seteaza dimensiunea cimpului curent la
               //valoarea 5
cout << i;
```

Valoarea lui *i* se afişează pe un cîmp de 5 caractere, cadrată în dreapta.

Primele două caractere ale cîmpului conţin spaţii. Acelaşi efect se obţine prin apelul de mai jos:

```
printf("%5d",i);
```

După o operaţie de inserare, dimensiunea cîmpului devine în mod automat cea implicită, adică egală cu zero.

Caracterele nesemnificative care însoţesc datele care se afişează se mai numesc şi *caractere de umplere*. Ele pot fi definite de utilizator cu ajutorul funcţiei membru *fill* a clasei *ios*.

Funcţia *fill* are două prototipuri:

```
char fill();
```

şi

```
char fill(char c);
```

Funcţia fără parametru returnează codul ASCII al caracterului de umplere curent.

Apelul funcţiei *fill* cu parametru conduce la definirea caracterului de umplere şi anume, caracterul de umplere devine egal cu cel al cărui cod ASCII este egal cu valoarea lui *c*. Funcţia returnează valoarea codului ASCII al vechiului caracter de umplere.

Exemplu:

```
int i=123;
...
cout.width(5);
cout.fill('0');
cout << i;
```

Se afişează:

00123

Acelaşi efect se obţine cu ajutorul apelului funcţiei *printf* de mai jos:

```
printf("%05d",i);
```

Funcţia *fill* oferă facilităţi superioare pentru a defini caracterele de umplere faţă de specificatorii de format utilizaţi la apelul lui *printf*. Într-adevăr, funcţia *fill* ne permite să utilizăm drept caractere de umplere şi alte caractere decît zero şi cel implicit, care sînt singurele caractere utilizabile folosind funcţia *printf*. De exemplu, apelul:

```
cout.fill('*');
```

defineşte drept caracter de umplere, caracterul asterisc.

Indicatorii de format enumeraţi mai sus, care definesc un atribut de format, sînt grupaţi în trei grupe. Aceste grupe pot fi referite prin denumiri atribuite lor. Primul grup de biţi formează grupa care defineşte cadrajul

datei în cîmpul de afişare. Ea se numeşte *adjustfield* şi este formată din biţii *right, left* şi *internal*. Aceşti biţi definesc poziţiile caracterelor de umplere: în stînga datei care se afişează (modul implicit), în dreapta datei sau în interiorul ei.

O altă grupă este denumită *basefield* şi ea conţine biţii *dec, oct* şi *hex*. Ea defineşte baza de numeraţie a datei care se afişează.

În sfirşit, cea de a treia grupă este denumită *floatfield* şi conţine biţii *scientific* şi *fixed* care definesc formatul de afişare al numerelor: cu exponent sau fără.

O caracteristică generală a acestor grupe este faptul că în cadrul fiecăreia, numai un bit poate fi setat. De exemplu, dacă este setat bitul *left*, atunci ceilalţi biţi din aceeaşi grupă nu mai pot fi setaţi.

Biţii indicatori de format pot fi setaţi folosind funcţia membru *setf* a clasei *ios*. Aceasta are două formate:

```
long setf(long f);
```

şi

```
long setf(long bit,long grupa);
```

Ambele funcţii returnează valoarea precedentă a dublu cuvîntului *x_flag* care defineşte formatul vechi.

Primul format al funcţiei *setf* modifică formatul curent setînd biţii setaţi în valoarea lui *f*. Ceilalţi biţi (care corespund biţilor avînd valoarea zero) ai formatului rămîn nemodificaţi.

Cel de al doilea format se utilizează pentru a seta biţii din cele trei grupe amintite mai sus: *adjustfield, basefield* şi *floatfield*. Primul parametru defineşte bitul din cadrul grupei, iar cel de al doilea grupa.

La apelurile acestei funcţii se pot utiliza enumeratorii şi denumirile celor trei grupe. Ei vor fi prefixaţi de numele *ios* urmat de operatorul de rezoluţie.

Exemple:

1.

```
int i = 123;
cout.width(5);
cout.setf(ios::left,ios::adjustfield);
cout.fill('0');
cout << i;
```

Se afişează:

12300

2.

```
int i = 123;
cout.width(5);
cout.setf(ios::right,ios::adjustfield);
cout.fill(' ');
cout.setf(ios::showpos);
cout << i;
```

Se afişează:

+123

caracterul + fiind precedat de caracterul spaţiu.

3.

```
int i = 123;
cout.width(5);
cout.setf(ios::showpos);
cout.fill('0');
cout.setf(ios::internal,ios::adjustfield);
cout << i;
```

Se afişează:

+0123

4.

```
int i = 123;
cout.setf(ios::hex,ios::basefield);
cout << i;
```

Se afişează:

7b

Se obţine acelaşi efect ca la apelul:

```
printf("%x",i);
```

5.

```
double d = 123.45;
cout << d;
```

Se afişează:

123.45

6.

```
double d = 123.45;
cout.setf(ios::scientific,ios::floatfield);
cout << d;
```

Se afişează:

1.2345e+02

Numărul de zecimale se poate defini cu ajutorul funcţiei membru *precision* a clasei *ios*. Ea are două prototipuri:

```
int precision();
```

şi

```
int precision(int n);
```

În primul caz, se returnează precizia curentă. În cel de al doilea caz, se returnează precizia existentă înaintea modificării. Precizia se modifică la *n* caractere.

Exemplu:

7.

```
double pi = 3.14159265;
cout.setf(ios::fixed,ios::floatfield);
cout.precision(7);
cout << pi;
```

Se afişează:

3.1415927

Exerciţii:

31.1 Să se scrie un program care afişează la display datele din exemplele 1-7 din paragraful 31.1.1.

PROGRAMUL BXXXI1

```
#include <iostream.h>
#include <conio.h>

main()
/* afiseaza date folosind ierarhiile de clase care implementeaza
   conceptul de stream */
{
```

```
int i=123;

// exemplul 1
cout.width(5);
cout.setf(ios::right,ios::adjustfield);
cout.fill('0');
cout  << i;
cout << "\n Actionati o tasta pentru a\
           continua\n";
getch();

// exemplul 2
cout.width(5);
cout.fill(' ');
cout.setf(ios::left,ios::adjustfield);
cout.setf(ios::showpos);
cout  << i;
cout << "\n Actionati o tasta pentru a\
           continua\n";
getch();

// exemplul 3
cout.width(5);
cout.setf(ios::showpos);
cout.fill('0');
cout.setf(ios::internal,ios::adjustfield);
cout  << i;
cout << "\n Actionati o tasta pentru a\
           continua\n";
getch();

// exemplul 4
cout.setf(ios::hex,ios::basefield);
cout  << i;
cout << "\n Actionati o tasta pentru a\
           continua\n";
getch();

// exemplul 5
double d = 123.45;
cout  << d;
cout << "\n Actionati o tasta pentru a\
           continua\n";
```

```
    getch();

    //exemplul 6
    cout.setf(ios::scientific,ios::floatfield);
    cout  << d;
    cout << "\n Actionati o tasta pentru a\
                continua\n";
    getch();

    //exemplul 7
    double pi = 3.14159265;
    cout.setf(ios::fixed,ios::floatfield);
    cout.precision(7);
    cout  << pi;
    cout << "\n Actionati o tasta pentru a\
                continua\n";
    getch();
}
```

31.1.1.1. Manipulatori

Manipulatorii permit definirea formatelor pentru operaţiile de intrare/ieşire. Practic, toate facilităţile de definire ale formatelor descrise mai sus, folosind funcţiile membru *width, setf, fill* etc., pot fi realizate cu ajutorul *manipulatorilor*. Aceştia sînt funcţii membru speciale. O astfel de funcţie returnează o referinţă la un stream. În felul acesta, apelurile manipulatorilor se pot înlănţui în inserări cu scopul de a defini biţii de format.

Indicăm mai jos numele, parametri şi acţiunea fiecărui manipulator.

Manipulator	Acţiune
dec	setează indicatorul de conversie în zecimal
oct	setează indicatorul de conversie în octal
hex	setează indicatorul de conversie în hexazecimal
ws	setează indicatorul skipws de salt peste caracterele albe
endl	inserează newline şi videază zona tampon a streamului

(continuare)

Manipulator	Acţiune
ends	inserează caracterul nul care termină şirurile de caractere
flush	videază zona tampon a unui obiect al clasei ostream
setbase(int *n*)	setează baza de conversie egală cu n; n poate avea una din valorile 0,8,10 sau 16; valoarea zero înseamnă conversie implicită, adică în zecimal
resetiosflags(long *a*)	şterge biţii de format specificaţi de parametrul a
setiosflags(long *a*)	setează biţii de format specificaţi de parametrul a
setfill(int *n*)	setează caracterul de umplere la valoarea lui n
setprecision(int *n*)	setează precizia la valoarea lui n
setw(int *n*)	setează dimensiunea cîmpului la valoarea lui n

Manipulatorii *dec, oct* şi *hex* definesc baza de conversie. Aceasta rămîne valabilă în continuare, pînă la o nouă modificare a ei.

Utilizarea manipulatorilor implică includerea fişierului *iomanip.h*.

Exerciţii:

31.2 Programul de faţă realizează aceleaşi afişări ca şi programul din exerciţiul 31.1, utilizînd manipulatori la definirea formatelor.

PROGRAMUL BXXXI2

```
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>

main()
/* afiseaza date folosind ierarhiile de clase care implementeaza
   conceptul de stream */
{
 int i=123;
```

```
// exemplul 1
cout << setw(5) <<
     resetiosflags(ios::internal|ios::right) <<
       setiosflags(ios::left) << setfill('0') << i;
cout << "\n Actionati o tasta pentru a\
              continua\n";
getch();

// exemplul 2
cout << setw(5) <<
     resetiosflags(ios::internal|ios::left) <<
       setiosflags(ios::right) << setfill(' ') <<
        setiosflags(ios::showpos) << i;
cout << "\n Actionati o tasta pentru a\
              continua\n";
getch();

// exemplul 3
cout << setw(5) <<
    resetiosflags(ios::left|ios::right) <<
      setiosflags(ios::internal|ios::showpos) <<
      setfill('0') << i;
cout << "\n Actionati o tasta pentru a\
              continua\n";
getch();

// exemplul 4
cout << hex << i;
cout << dec;   // se reface conversia in zecimal
cout << "\n Actionati o tasta pentru a\
              continua\n";
getch();

// exemplul 5
double d = 123.45;
cout  << d;
cout << "\n Actionati o tasta pentru a\
              continua\n";
getch();

// exemplul 6
cout << resetiosflags(ios::fixed) <<
     setiosflags(ios::scientific) << d;
```

```
cout << "\n Actionati o tasta pentru a\
          continua\n";
getch();

//exemplul 7
double pi = 3.14159265;
cout << setprecision(7) << pi;
cout << "\n Actionati o tasta pentru a\
          continua\n";
getch();
}
```

31.1.1.2. Ieşire neformatată

Ieşirea unui singur caracter se poate realiza cu ajutorul funcţiei membru *put* a clasei *ostream*. Aceasta are prototipul:

```
ostream& ostream::put(char c);
```

Apelul:

```
char c='A';
cout.put(c);
```

are acelaşi efect cu instrucţiunea expresie:

```
cout << c;
```

dacă biţii de format au valori implicite.

Pentru a afişa *n* octeţi, se poate utiliza funcţia membru *write* a clasei *ostream*. Ea are următoarele prototipuri:

ostream& write(const signed char *sir,int n);

şi

ostream& write(const unsigned char *sir,int n);

Funcţia *write* afişează *n* caractere chiar dacă printre cei *n* octeţi spre care pointează *sir* se întîlneşte caracterul *nul*.

Esenţa acestor funcţii este faptul că, asupra lor biţii de format nu au nici un efect.

31.1.1.3. Supraîncărcarea operatorului << pentru ieşiri de obiecte

Utilizarea operatorului de inserare pentru tipuri abstracte este posibil dacă acesta se supraîncarcă în mod corespunzător.

De exemplu, operatorul << se utilizează în expresii de forma:

(1) **stream <<** *data_de_tip_predefinit*

El poate fi supraîncărcat pentru a fi utilizat cu obiecte care sînt instanţieri ale unei clase. Cu alte cuvinte, operatorul de inserare se supraîncarcă aşa încît expresia:

(2) **stream <<** *obiect*

să fie acceptată de compilator.

Operandul *stream* este o instanţiere a clasei *ostream* şi de aici rezultă că operatorul << poate fi supraîncărcat pentru o clasă oarecare numai printr-o funcţie prieten. Din expresia (2), rezultă că funcţia care supraîncarcă operatorul de inserare are doi parametri: primul este o referinţă la clasa *ostream*, iar cel de al doilea parametru este un obiect al cărui tip este tipul abstract pentru care se face supraîncărcarea operatorului respectiv.

Deci, prototipul funcţiei care supraîncarcă operatorul de inserare pentru clasa C este următorul:

friend ostream& *operator* **<< (ostream&,***C***);**

Funcţia returnează o referinţă la clasa *ostream* pentru ca operatorul de inserare să se poată înlănţui ca în cazul operanzilor de tipuri predefinite.

De exemplu, dacă clasa C este clasa care implementează conceptul de număr *complex*, atunci prototipul funcţiei care supraîncarcă operatorul de inserare va fi:

friend ostream& *operator* **<< (ostream&,***complex***);**

Această funcţie o putem defini astfel:

```
ostream& operator << (ostream& iesire,complex z)
// supraincarca operatorul de inserare
{
 iesire << z.real << "+(" << z.imag <<")i";
 return iesire;
}
```

Clasa *complex* a fost definită în mai multe exerciţii din cartea de faţă. O parte din elementele clasei *complex* se definesc ca mai jos:

```
class complex {
    double real;
    double imag;
 public:
    complex(double x,double y)
    {
```

```
        real = x;
        imag = y;
    }

    friend ostream& operator << (ostream&,
                                 complex);
    ...
};
```

Fie instanţierea:

```
complex a(1,2);
```

Instrucţiunea expresie:

```
cout << "a=" << a << "\n";
```

afişează:

a = 1+(2)i

31.1.2. Intrare standard

Intrările standard se pot realiza folosind operatorul ">>". Acesta este supraîncărcat pentru intrările standard. Operandul stîng este un obiect al clasei *istream*. Operandul din dreapta are un tip pentru care a fost supraîncărcat operatorul ">>". Acesta poate fi un tip predefinit sau un tip abstract.

Operatorul ">>" supraîncărcat pentru operaţii de intrare se numeşte *operator de extragere* (extraction) sau *obţine de la...* (get from).

În mod implicit, operatorul de extragere este supraîncărcat pentru tipurile predefinite şi pentru şiruri de caractere.

Amintim că operatorul ">>" este, în mod normal, operatorul de deplasare la dreapta. El are aceeaşi prioritate şi asociativitate ca şi operatorul "<<".

Funcţia care supraîncarcă operatorul de extragere returnează o referinţă la obiectul curent şi de aceea, operatorii de extragere se pot aplica înlănţuiţi.

Prototipul funcţiei membru a clasei *istream* care supraîncarcă operatorul de *extragere* este următorul:

```
istream& operator >> (tip& );
```

În mod implicit, operatorul de extragere realizează avansul peste caracterele albe pînă la primul caracter care nu este alb.

Utilizatorul poate să suprime acest avans prin ştergerea bitului *ios::skipws*. În acest scop se poate folosi manipulatorul *resetiosflags* (vezi paragraful 31.1.1.1.) sau funcţia *setf* (vezi paragraful 31.1.1.).

Caracterele cîmpului curent trebuie să corespundă tipului de dată care formează operandul drept al operatorului de extragere. În caz contrar, se intră în starea de eroare.

Exemple:

1.

```
int i;
...
cin >> i;
```

Această instrucţiune expresie are acelaşi efect cu apelul:

```
scanf("%d",&i);
```

2.

```
double d;
...
cin >> d;
```

Această instrucţiune expresie are acelaşi efect cu apelul:

```
scanf("%lf",&d);
```

3.

```
char c;
...
cin >> c;
```

Această instrucţiune expresie citeşte primul caracter care nu este alb (dacă bitul ios::skipws nu este în prealabil şters); are acelaşi efect cu secvenţa:

```
scanf("%1s",tc);
```

unde *tc* este un tablou de tip *char* de cel puţin 2 caractere:

```
char tc[2];
```

4.

```
char tab[100];
cin >> tab;
```

Această instrucţiune expresie are acelaşi efect cu apelul:

```
scanf("%s",tab);
```

În legătură cu şirurile de caractere, poate apare situaţia cînd la intrare se află un şir de caractere care depăşeşte dimensiunea tabloului destinat pentru a păstra caracterele citite. Pentru a preîntîmpina acest lucru, se poate defini dimensiunea cîmpului egală cu a zonei receptoare a şirului de caractere care se citeşte. În acest scop, se va folosi funcţia *width* (vezi paragraful 31.1.1.) într-o secvenţă de forma:

```
char tab[...];
cin.width(sizeof tab);
cin >> tab;
```

În mod implicit, cîmpul din care se face transferul începe la primul caracter care nu este alb şi se termină la primul caracter care nu aparţine tipului operatorului de extragere sau la întîlnirea unui caracter alb.

Utilizatorul poate defini dimensiunea maximă a cîmpului cu ajutorul funcţiei *width*, ca în exemplul de mai sus.

În cazul operatorului de extragere se pot folosi aproape toţi manipulatorii definiţi în paragraful 31.1.1.1.

În particular, manipulatorul *ws* se poate folosi numai în operaţii de *extragere*. El permite activarea sau dezactivarea saltului peste caracterele albe.

Manipulatorii care nu pot fi utilizaţi în operaţii de extragere decît numai în operaţii de inserare sînt:

endl, ends, flush şi *setbase*.

Un *stream* are asociat o *stare de eroare* care se defineşte cu ajutorul datei membru *state* a clasei *ios*. Biţii acestei date au semnificaţia indicată mai jos:

```
class ios {
 public:
    ...
    enum io_state {
        goodbit = 0x00,   // operatia de intrare/iesire corecta
                          // daca acest bit este pozitionat
                          // ceilalti sint stersi
        eofbit=0x01,      // s-a intilnit sfirsitul de fisier intr-o
                          // operatie de intrare
        failbit=0x02,     // se seteaza daca ultima operatie de
                          // intrare/iesire (extragere sau conversie)
                          // a esuat; streamul este utilizabil in
                          // continuare daca se sterge bitul de
                          // eroare
```

```
            badbit=0x04,     // se seteaza daca ultima operatie de
                             // intrare/iesire a fost invalida; streamul
                             // poate fi utilizat in continuare dupa ce
                             // se sterge conditia de eroare
            hardfail=0x08    // se seteaza la o eroare irecuperabila
        };
        ...
    };
```

Un *stream* intrat într-o stare de eroare nu mai permite operaţii de intrare/ieşire (inserare sau extragere) pînă cînd condiţia de eroare este înlăturată şi biţii de eroare sînt şterşi.

Accesul la biţii de eroare a datei *state* se obţine folosind următoarele funcţii:

int good(); - Returnează o valoare diferită de zero dacă operaţia de intrare/ieşire s-a efectuat fără erori (bitul *goodbit* este setat).

int eof(); - Returnează o valoare diferită de zero dacă s-a intîlnit sfîrşitul de fişier (bitul *eofbit* este setat).

int fail(); - Returnează o valoare diferită de zero dacă este poziţionat cel puţin unul din biţii *failbit* sau *hardfail*.

int bad(); - Returnează o valoare diferită de zero dacă este poziţionat bitul *badbit*.

Valoarea datei membru *state* este returnată de funcţia:

int rdstate(); - Biţii de eroare pot fi modificaţi folosind funcţia membru *clear* a clasei *ios*.

Această funcţie are prototipul:

```
void clear(int i= 0);
```

Ea permite setarea biţilor de eroare.

De exemplu, pentru a seta bitul *failbit* vom utiliza construcţia:

(1) `ios::failbit`

Pentru ca ceilalţi biţi să rămînă pe loc, apelăm funcţia *rdstate* pentru streamul curent, de exemplu, pentru *cin*:

(2) `cin.rdstate()`

Bitul *failbit* se setează apelînd funcţia *clear* ca mai jos:

```
cin.clear(ios::failbit|cin.rdstate());
```

Un apel, fără parametri, al funcţiei *clear* permite anularea tuturor biţilor de eroare, cu excepţia bitului *hardfail* care nu poate fi anulat.

Un stream poate fi testat pentru a stabili dacă se află sau nu într-o stare de eroare, la fel ca o expresie logică.

Aceasta este posibil deoarece clasa *ios* are supraîncărcat operatorul "!", precum şi conversia unui *stream* într-un pointer spre *void*.

Funcţia membru care supraîncarcă operatorul "!" are prototipul:

```
int operator !();
```

Acest operator poate fi utilizat în expresii de forma:

```
if(!stream)
...
```

Expresia *!stream* are o valoare diferită de zero (adevărat), dacă cel puţin unul din biţii de eroare ai stării lui *stream* este setat; altfel expresia are valoarea zero (fals).

Funcţia de conversie a streamului într-un pointer spre *void* are prototipul:

```
operator void *();
```

Conversia streamului într-un pointer spre *void* are ca rezultat pe zero (pointerul nul) dacă cel puţin unul din biţii de eroare ai stării streamului este setat şi o valoare diferită de zero în caz contrar. Menţionăm că pointerul rezultat printr-o astfel de conversie se poate folosi numai pentru a testa dacă biţii de eroare ai stării streamului sînt setaţi sau nu.

O expresie de forma:

```
cin >> v
```

are ca valoare o referinţă la streamul *cin*, de tip *istream*. O astfel de referinţă poate fi utilizată într-o instrucţiune de forma:

```
if(cin >> v)
...
```

În acest caz, se aplică automat conversia referinţei respective spre un pointer spre *void*. Rezultatul acestei conversii este pointerul nul dacă cel puţin un bit de eroare al stării streamului *cin* este setat.

În caz contrar, pointerul rezultat este nenul.

Exerciţii:

31.3 Să se scrie o funcţie care realizează următoarele:

– afişează un text la terminalul standard;

– citeşte un întreg de tip *int* de la intrarea standard;

– returnează valoarea zero dacă se întîlneşte sfîrşitul de fişier şi unu în caz contrar.

Funcţia are doi parametri:

text - Este pointerul spre şirul de caractere care se afişează la terminal.

n - Este referinţă la întregi de tip *int*.

Funcţia păstrează întregul citit în zona de memorie referită de *n*.

FUNCŢIA BXXXI3

```
#ifndef __IOSTREAM_H
#include <iostream.h>
#define __IOSTREAM_H
#endif

int citInt(char *text,int& n)
/* - afiseaza text;
   - citeste un intreg de tip int si-l pastreaza in zona referita de n;
   - returneaza zero la sfirsitul de fisier si unu altfel.
*/
{
 char t[255];

 for(;;) {
     cout << text;
     if(cin >> n)
         return 1;    // citire fara eroare
     if(cin.eof())
         return 0;    // s-a intilnit EOF

     // sterge starea de eroare
     cin.clear();

     // videaza buferul
     cin >> t;

     cout << "Nu s-a tastat un intreg\n";
 }
}
```

31.4 Să se scrie un program care citeşte, de la intrarea standard, un şir de numere de tip *int* şi afişează suma lor. După ultimul număr al şirului se tastează sfîrşitul de fişier.

PROGRAMUL BXXXI4

```
#include <conio.h>
#include <stdio.h>
#include "BXXXI3.CPP"
main()
/* citeste numere de tip int si afiseaza suma lor */
{
 int s = 0;
 int i = 1;
 int nr;

 for(;;) {
    char zona[20];
    sprintf(zona,"n%d=",i);
    if(citInt(zona,nr) == 0)
      break;
    s += nr;
    i++;
 }
 cout << "S-au citit " << i-1 << " intregi\n";
 cout << "Suma lor este = " << s << "\n";
}
```

Observaţie:

După modelul funcţiei *citInt*, se pot construi funcţii care să permită citiri de numere de tip *long*, *float*, *double* sau *long double*. Propunem cititorului să realizeze singur astfel de funcţii. Pentru testarea lor se pot folosi programe de felul celui de mai sus.

Instrucţiunea expresie:

```
cin >> t;
```

utilizată în funcţia *citInt* pentru a vida buferul, extrage din obiectul *cin* caractere din cîmpul care începe cu caracterul eronat (care nu corespunde unui întreg) şi pînă la întîlnirea unui caracter alb. Dacă după acest caracter alb urmează şi alte caractere, atunci instrucţiunea de mai sus nu este suficientă pentru vidarea buferului de intrare. O secvenţă eficientă în toate situaţiile se indică în paragraful 31.2.

31.1.2.1. Intrări neformatate

Intrarea unui singur caracter se poate realiza cu ajutorul funcţiei membru *get* a clasei *istream*. Ea are prototipul:

istream& get(char&);

Se obişnuieşte să se spună că funcţia *get* extrage un caracter din streamul curent şi-l păstrează în zona de memorie referită de parametrul formal.

Funcţia *get* nu realizează avansul peste caracterele albe, deci ea extrage din streamul curent caracterul, indiferent că acesta este un spaţiu alb sau nu.

Funcţia *get* are şi alte prototipuri care permit extragerea unui şir de caractere din streamul curent şi anume:

istream& get(unsigned char ***zona*,int** *max***,int** *term* **= '***n***');**
istream& get(signed char ***zona*,int** *max***,int** *term* **= '***n***');**

Această funcţie extrage din streamul curent cel mult *max-1* caractere sau pînă la întîlnirea caracterului dat de valoarea lui *term*. Caracterele respective se păstrează în zona de memorie spre care pointează *zona*.

După caracterele extrase din streamul curent se păstrează caracterul NUL ('\0'). De aceea, zona de memorie spre care pointează *zona* trebuie să conţină cel puţin *max* octeţi.

Terminatorul definit de *term* nu se păstrează în zona de memorie spre care pointează *zona*. De altfel, dacă extragerea se termină la întîlnirea lui, acesta nici nu este extras din streamul curent.

Extragerea caracterului definit de *term* din streamul de intrare este posibilă cu ajutorul funcţiei *getline*.

Aceasta are un prototip similar cu funcţia *get*:

istream& getline(char ***zona*,int** *max***,int** *term* **= '***n***');**

Parametrii formali ai funcţiei *getline* au aceleaşi sens ca în cazul funcţiei *get*.

Terminatorul definit de *term* nu se transferă în zona de ieşire ci numai se extrage din streamul de intrare.

Extragerea binară a datelor din streamul curent se realizează folosind funcţia *read*. Aceasta are prototipurile:

istream& read(unsigned char ***zona*,int** *n***);**
istream& read(signed char ***zona*,int** *n***);**

Funcţia extrage *n* octeţi din streamul curent şi îi transferă în zona de memorie spre care pointează *zona*.

Exemple:

1.

```
char c;
cin.get(c);
```

Are acelaşi efect cu apelul:

```
c=getchar();
```

2.

```
char tab[100];
cin.get(tab,100,'\n');
```

Citeşte cel mult 99 de caractere de la intrarea standard pe care le păstrează în tabloul *tab*. Dacă se întîlneşte caracterul '\n' înainte de a citi 99 de caractere, citirea se întrerupe. Caracterul '\n' rămîne în zona tampon a streamului curent.

După ultimul caracter transferat din streamul curent se memorează caracterul NUL.

O funcţie membru a clasei *istream* care se utilizează frecvent este funcţia *putback*. Aceasta permite punerea unui caracter într-un stream de intrare. Streamul intră în starea "fail" dacă nu se poate pune caracterul respectiv în stream.

Funcţia *putback* are prototipul:

istream& putback(char *c*);

Un apel de forma:

```
cin.putback(c);
```

pune caracterul *c* în streamul *cin*.

Extragerea referitoare la *cin*, care urmează acestui apel, va începe cu caracterul *c*.

Funcţia *putback* îşi găseşte utilizări în programe de *analiză lexicală* care recunosc diferite tipuri de date aflate la intrare. De exemplu, dacă la intrare se află o expresie de forma:

a1b2+x

atunci analiza lexicală a acesteia trebuie să recunoască următoarele tipuri de date:

- a1b2 identificator;

– +   caracter special;

– x   identificator.

În general, prin *identificator* se înţelege o succesiune de litere şi cifre care încep cu o literă. Pentru a recunoaşte un identificator vom proceda la citirea ciclică a caracterelor de la intrare cît timp acestea sînt litere sau cifre. La întîlnirea unui caracter care nu este nici literă şi nici cifră, înseamnă că s-a "avansat" peste caracterele identificatorului, citindu-se chiar şi caracterul care urmează după identificator. De obicei, acest caracter trebuie pus înapoi, pentru a fi recitit şi recunoscut ulterior. În caz contrar, el poate fi pierdut. De exemplu, în cazul de mai sus, după citirea caracterelor identificatorului *a1b2* se citeşte caracterul "+". Abia în acest moment se ştie că identificatorul respectiv s-a terminat. Urmează repunerea caracterului "+" în streamul de intrare pentru a putea fi ulterior recitit şi tratat.

Ţinînd seama de cele de mai sus, putem construi o funcţie pentru recunoaşterea identificatorilor ca mai jos:

```
void ident(char *id)
  /* citeste caracterele identificatorului curent si le pastreaza in zona spre
     care pointeaza id */
  {
   *id = '\0';
   char car;
   car = 0;

   cin >> car; //salt peste spatii albe
   if(car >= 'A' && car <= 'Z' || car >= 'a' &&
      car <= 'z')
      //litera
      do{
          *id++ = car;   //pastreaza caracterul curent al
                         //identificatorului
          car = 0;
          cin.get(car);
          int i = car >= 'A' && car <= 'Z' ||
                  car >= 'a' && car <= 'a' ||
                  car >= '0' && car <= '9';
          /* i=1 pentru cazul cind car este litera sau cifra;
             i=0 altfel. */
      } while(i);
      *id = '\0';   //sfirsitul identificatorului
```

```
    /* car contine un caracter care nu este nici litera si nici cifra;
       acest caracter este urmator identificatorului extras din streamul
       cin; acesta trebuie repus in streamul cin pentru a putea fi
       recitit ulterior */

  cin.putback(car);
}
```

31.1.2.2. Supraîncărcarea operatorului >> pentru intrări de obiecte

Utilizarea operatorului de extragere pentru tipuri abstracte este posibilă dacă acesta se supraîncarcă în mod corespunzător.

În cazul tipurilor predefinite, operatorul de extragere a fost folosit în expresii de forma:

(1) **cin >>** *data_de_tip_predefinit*

De aceea, alături de expresia (1), se doreşte acceptarea de către compilator a expresiilor de forma:

(2) **cin >>** *obiect*

unde *obiect* este o instanţiere a unei clase oarecare CL. Acest lucru este posibil dacă se supraîncarcă operatorul ">>" în mod corespunzător pentru clasa CL. O astfel de supraîncărcare este posibilă printr-o funcţie prieten a clasei CL. Antetul funcţiei va fi următorul:

istream& *operator* **>> (istream&,***CL***&);**

Funcţia returnează o referinţă la clasa *istream* ceea ce permite utilizarea înlănţuită a operatorului de *extragere*.

De exemplu, dacă *CL* este clasa care implementează conceptul de număr complex, atunci prototipul funcţiei care supraîncarcă operatorul de extragere va fi:

friend istream& *operator* **>> (istream&,complex&);**

Exerciţii:

31.5 Să se implementeze tipul abstract *epunct* pentru gestiunea punctelor pe ecranul grafic.

FIŞIERUL BXXXI5.H

```
#ifndef __GRAPHIC_H
#include <graphics.h>
#define __GRAPHIC_H
#endif
```

```
#ifndef __IOSTREAM_H
#include <iostream.h>
#define __IOSTREAM_H
#endif

class epunct {
 protected:
     int x;
     int y;
     int valid;   /* valid=1 daca punctul se afla in limitele
                                         ecranului grafic;
                         altfel valid=0 */
     int culoare;
     int vizibil;

     void testecran();
     /* defineste valoarea variabilei valid pentru obiectul curent */
 public:
     epunct();    // constructor implicit
     epunct(int abs,int ord=0,int c=WHITE);
     // constructor pentru initializari si conversii

     epunct(const epunct&);   // constructor de copiere
     epunct& operator = (const epunct&);
     // supraincarca operatorul de atribuire

     friend istream& operator >>
           (istream&,epunct&);
     // supraincarca operatorul de extragere

     void afiseaza();
     // afiseaza punctul curent

     void sterge();
     // face invizibil punctul curent

     int retabs() const;
     // returneaza abscisa punctului curent

     int retord() const;
     // returneaza ordonata punctului curent
```

```
    int retculoare() const;
    // returneaza codul culorii punctului curent

    int retvalid() const;
    // returneaza valoarea lui valid
};
```

Funcţiile membru se definesc în fişierul de mai jos, de extensie *CPP*.

FIŞIERUL BXXXI5

```
#ifndef __EPUNCT_H
#include "BXXXI5.H"
#define __EPUNCT_H
#endif

void epunct::testecran()
/* atribuie 1 variabilei valid daca x si y definesc un punct
   in limitele ecranului grafic; altfel valid = 0 */
{
 if(x < 0 || x > getmaxx() || y < 0 ||
    y > getmaxy())
    valid = 0;
 else
    valid = 1;
}

inline epunct::epunct() // constructor implicit
{
 valid = vizibil = 0;
}

epunct::epunct(int abs,int ord,int c)
// constructor pentru initializari si conversii
{
 x = abs;
 y = ord;
 culoare = c;
 vizibil = 0;
 testecran();
}

epunct::epunct(const epunct& ep) // constructor de copiere
{
 x = ep.x;
```

```
 y = ep.y;
 culoare = ep.culoare;
 vizibil = ep.vizibil;
 valid = ep.valid;
}

epunct& epunct::operator = (const epunct& ep)
// supraincarca operatorul =
{
 if(ep.valid && this != &ep){
     x = ep.x;
     y = ep.y;
     culoare = ep.culoare;
     vizibil = ep.vizibil;
     valid = ep.valid;
 }
 else
     if(ep.valid == 0)
         valid = 0;
 return *this;
}

istream& operator >> (istream& intrare,epunct& ep)
// supraincarca operatorul de extragere
{
 int a,b,c;
 char t[255];
 char er[] = "Nu s-a tastat un intreg\n";
 static istream *pnul = 0;

 for(;;) {
     for(;;) {    // citeste abscisa
         cout << "abscisa=";
         if(intrare >> a)
             break;
         if(intrare.eof())
             return *pnul;   // s-a intilnit EOF
         cout << er;
         intrare.clear();   // sterge starea de eroare
         intrare.getline(t,255,'\n');
         // videaza streamul de intrare
     }
```

```
    //citeste ordonata
    for(;;) {
        cout << "ordonata=";
        if(intrare >> b)
            break;
        if(intrare.eof())
            return *pnul;   //s-a intilnit EOF
        cout << er;
        intrare.clear();   //sterge starea de eroare
        intrare.getline(t,255,'\n');
        //videaza streamul de intrare
    }
    epunct crt(a,b);
    if(crt.valid)
        break;   //crt este in limitele ecranului
    cout << "coordonatele tastate nu sint in\
            limitele ecranului\n";
    cout << "se reia citirea coordonatelor\n";
  }

  for(;;) {   //citeste codul pentru culoarea punctului
    cout << "culoare=";
    if((intrare >> c) && c >= 0 && c <= 15)
        break;
    if(intrare.eof())
        return *pnul;  //s-a intilnit EOF
    intrare.clear();   //sterge starea de eroare
    cout << "se cere intreg in intervalul\
            [0,15]\n";
    intrare.getline(t,255,'\n');
  }
  ep.x = a;
  ep.y = b;
  ep.culoare = c;
  ep.valid = 1;
  ep.vizibil = 0;
  return intrare;
}

inline void epunct::afiseaza()
//afiseaza pe ecranul grafic punctul curent
{
  if(valid){
```

```
    putpixel(x,y,culoare);
    vizibil = 1;
 }
}

inline void epunct::sterge()
// face invizibil punctul curent
{
 if(valid)
    putpixel(x,y,getbkcolor());
 vizibil = 0;
}

inline int epunct::retabs() const
// returneaza abscisa punctului curent
{
 return x;
}

inline int epunct::retord() const
// returneaza ordonata punctului curent
{
 return y;
}

inline int epunct::retculoare() const
// returneaza codul culorii punctului curent
{
 return culoare;
}

inline int epunct::retvalid() const
{
 return valid;
}
```

31.6 Să se implementeze tipul abstract *tabpct* care gestionează, în memoria *heap*, un tablou de obiecte ale clasei *epunct* definită în exercițiul 31.5.

Clasa *tabpct* are următoarele date membru:

- *n* — numărul obiectelor de tip *epunct* păstrate într-o zonă contiguă din memoria *heap*;
- *tab* — pointer spre zona din memoria *heap* în care se păs-

trează cele *n* obiecte de tip *epunct*;

- *vtabpct* — variabilă de tip *int* care are valoarea 1 dacă cele *n* obiecte din memoria *heap* sînt afişate şi zero în caz contrar.

FIŞIERUL BXXXI6.H

```
#ifndef __EPUNCT_H
#include "BXXXI5.CPP"
#define __EPUNCT_H
#endif

class tabpct {
 protected:
     epunct *tab;
     int n;
     int vtabpct;
 public:
     tabpct();
     tabpct(int nr);
     tabpct(const tabpct&);
     ~tabpct();
     tabpct& operator = (const tabpct&);
     void afiseaza();
     // afiseaza punctele din zona spre care pointeaza tab

     void sterge();
     // face invizibile punctele din zona spre care pointeaza tab
};
```

Funcţiile membru se definesc în fişierul de mai jos, de extensie *CPP*.

FIŞIERUL BXXXI6

```
#ifndef __TABPCT_H
#include "BXXXI6.H"
#define __TABPCT_H
#endif

tabpct::tabpct()
{
 n=0;
}
```

```
tabpct::tabpct(int nr)
{
 vtabpct = 0;
 tab = 0;

 if(nr < 1) {
    cout << "numar puncte = " << nr <<
            " eronat\n";
    n = 0;
 }
 else {
    n = nr;

    // se rezerva zona de memorie in memoria heap
    tab = new epunct[n];

    if(tab == 0) {
       cout << "Memorie insuficienta\n";
       n = 0;
    }
    else{      // se citesc coordonatele punctelor
       for(int i=0;i < n;i++)
          if(!(cin >> tab[i]))
             break;
       if(i < n){
          cout << "S-au citit " << i <<
                   " puncte in loc de " << n <<
                   "\n";
          n = i;
          cin.clear();
       }
    }  // sfirsit else
 }
} // sfirsit constructor

tabpct::tabpct(const tabpct& tp)
{
 n = tp.n;
 vtabpct = tp.vtabpct;
 tab = new epunct[n];

 if(tab == 0){
    cout << "Memorie insuficienta\n";
```

```
        n = 0;
    }
    else
        for(int i=0; i < n;i++)
            tab[i] = tp.tab[i];
}

tabpct& tabpct::operator = (const tabpct& tp)
{
    if(this != &tp){
        n = tp.n;
        vtabpct = tp.vtabpct;
        if(n > 0)
            delete tab;
        tab = new epunct[n];

        if(tab == 0){
            cout << "Memorie insuficienta\n";
            n = 0;
        }
        else
            for(int i=0;i < n;i++)
                tab[i] = tp.tab[i];
    }
    return *this;
}

inline tabpct::~tabpct()
{
    delete tab;
}

void tabpct::afiseaza()
// afiseaza pe ecranul grafic punctele pastrate in zona spre
// care pointeaza tab
{
    for(int i=0;i < n;i++)
        tab[i].afiseaza();
    vtabpct = 1;
}
```

```
void tabpct::sterge()
// face invizibil punctele pastrate in zona spre care pointeaza tab
{
 for(int i=0;i < n;i++)
    tab[i].sterge();
 vtabpct = 0;
}
```

31.7 Să se scrie un program care trasează graficul unei funcţii care se defineşte tabelar.

PROGRAMUL BXXXI7

```
#include <stdlib.h>
#include <conio.h>
#include "BXXXI6.CPP"
#include "BXXXI3.CPP"

main() /* traseaza graficul unei functii date prin puncte */
{
 int nrpuncte = 0;
 if(citInt("numarul punctelor =",nrpuncte) == 0){
    cout << " s-a tastat EOF\n";
    exit(1);
 }
 if(nrpuncte == 0){
    cout << "numarul de puncte = 0\n";
    exit(1);
 }
 int gd = DETECT,gm;
 initgraph(&gd,&gm,"C:\\borlandc\\bgi");
 tabpct functie(nrpuncte);
 cleardevice();
 functie.afiseaza();
 getch();
 closegraph();
}
```

Observaţie:

Pentru a obţine un grafic care să aibă un aspect apropiat de cel real, este nevoie să se utilizeze coeficienţii de aspect.

31.8 Punctele afişate cu ajutorul funcţiei *putpixel* au o vizibilitate redusă datorită dimensiunii mici. Uneori se preferă afişarea unor puncte de dimensiuni mai mari pentru ca acestea să iasă mai bine în evidenţă pe

ecranul grafic. În acest scop, se poate afişa punctul respectiv, împreună cu un cerc, cu centrul în acel punct şi de rază trei. În acest exerciţiu se completează clasa *epunct*, definită în exerciţiul 31.6, cu funcţiile membru *cafiseaza* şi *csterge*, care afişează şi respectiv face invizibil un punct de pe ecranul grafic împreună cu cercul cu centrul în acel punct şi de rază trei.

În cele ce urmează vom numi *c_punct* imaginea unui punct pe ecranul grafic împreună cu a cercului cu centrul în punctul respectiv şi de rază trei.

FIŞIERUL BXXXI8.H

```
#ifndef __EPUNCT_H
#include "BXXXI5.CPP"  //clasa epunct
#define __EPUNCT_H
#endif

class cepunct:public epunct {
 public:
    void cafiseaza();
    //afiseaza obiectul curent ca un c_punct

    void csterge();
    //face invizibil obiectul curent care este afisat ca un c_punct

    cepunct();
    cepunct(int abs,int ord,int c=WHITE);
};
```

Funcţiile membru se definesc în fişierul de mai jos, de extensie *CPP*.

FIŞIERUL BXXXI8

```
#ifndef __CEPUNCT_H
#include "BXXXI8.H"
#define __CEPUNCT_H
#endif

inline cepunct::cepunct():epunct()
{
}

inline cepunct::cepunct(int abs,int ord,int c):
      epunct(abs,ord,c)
{
}
```

```
void cepunct::cafiseaza()
// afiseaza obiectul curent ca un c_punct
{
 int i,j;

 if(valid) {
    getaspectratio(&i,&j);
    double f = (double) i/j;
    putpixel(x,f*y,culoare);
    int temp = getcolor();
    setcolor(culoare);
    circle(x,f*y,3);    // traseaza un cerc de raza 3
    setcolor(temp);
    vizibil = 1;
 }
}

void cepunct::csterge()
// face invizibil obiectul curent care este considerat ca
// este afisat ca un c_punct
{
 if(valid) {
    int temp = culoare;
    culoare = getbkcolor();
    cafiseaza();
    culoare = temp;
    vizibil = 0;
 }
}
```

Observaţii:

1. Funcţia *cafiseaza* utilizează coeficienţii de aspect pentru a obţine imagini pe ecranul grafic apropiate de cele reale.
2. Acest exemplu ilustrează avantajul conceptului de moştenire: modificările se obţin simplu prin adăugări de facilităţi noi, nu prin schimbarea şi reprogramarea conceptelor existente.

Moştenirea este un concept care încurajează programarea prin componente standardizate care se bucură de o mare extensibilitate.

Se pot crea biblioteci de componente standardizate care se pot utiliza în programe ca atare. Dacă sînt necesare modificări, atunci acestea se fac nu modificînd componentele respective, ci prin extensia lor, adăugîndu-se facilităţi noi pe baza conceptului de moştenire. În felul acesta se măreşte

eficienţa în programare pe baza reutilizării simple a componentelor deja existente şi testate.

31.9 Să se extindă tipul abstract *tabpct*, implementat în exerciţiul 31.6, cu funcţii pentru afişarea şi ştergerea *c_punctelor* (pentru noţiunea de c_punct vezi exerciţiul 31.8).

FIŞIERUL BXXXI9.H

```
#ifndef __CEPUNCT_H
#include "BXXXI8.CPP"  //clasa cepunct
#define __CEPUNCT_H
#endif
#ifndef __TABPCT_H
#include "BXXXI6.CPP"   //clasa tabpct
#define __TABPCT_H
#endif

class ctabpct:public tabpct {
 protected:
     cepunct *ctab;
 public:
     ctabpct();
     ctabpct(int nr);
     void cafiseaza();
     //afiseaza punctele din zona spre care pointeaza ctab;
     //punctele se afiseaza ca niste c_puncte

     void csterge();
     //face invizibile punctele din zona spre care pointeaza ctab;
     //se considera ca punctele respective sint afisate ca niste
     //c_puncte
};
```

Mai jos, se definesc funcţiile membru într-un fişier de extensie *CPP*.

FIŞIERUL BXXXI9

```
#ifndef __CTABPCT_H
#include "BXXXI9.H"
#define __CTABPCT_H
#endif
```

```
inline ctabpct::ctabpct()
{
 n=0;
}

inline ctabpct::ctabpct(int nr):tabpct(nr)
{
 (epunct *)ctab = tab;
}

void ctabpct::cafiseaza()
/* afiseaza punctele din zona spre care pointeaza ctab;
   punctele se afiseaza ca niste c_puncte */
{
 for(int i = 0;i < n;i++)
     ctab[i].cafiseaza();
 vtabpct = 1;
}

void ctabpct::csterge()
/* face invizibile punctele din zona spre care pointeaza ctab;
   se considera ca punctele respective sint afisate ca niste c_puncte */
{
 for(int i = 0;i < n;i++)
     ctab[i].csterge();
 vtabpct = 0;
}
```

31.10 Să se implementeze tipul abstract *graficpct* pentru trasarea graficelor funcţiilor date prin puncte.

Prin graficul unei funcţii dată prin puncte vom înţelege imaginea de pe ecranul grafic compusă din:

- imaginile punctelor date sub formă de c_puncte;
- linii drepte care unesc punctele consecutive date ale funcţiei.

Liniile au caracteristicile standard (SOLID_LINE, NORM_WIDTH), iar culoarea coincide cu a primului punct dintre cele două puncte consecutive.

FIŞIERUL BXXXI10.H

```
#ifndef __CTABPCT_H
#include "BXXXI9.CPP"
```

```
#define __CTABPCT_H
#endif

class graficpct:public ctabpct {
 public:
    graficpct();
    graficpct(int nr);
    void afisgrafic(); //afiseaza graficul prin puncte
    void stergegrafic();
   // face invizibil graficul prin puncte
};
```

Funcţiile membru se definesc în fişierul de mai jos, de extensie *CPP*.

FIŞIERUL BXXXI10

```
#ifndef __GRAFICPCT_H
#include "BXXXI10.H"
#define __GRAFICPCT_H
#endif

inline graficpct::graficpct():ctabpct()
{
}

inline graficpct::graficpct(int nr):ctabpct(nr)
{
}

void graficpct::afisgrafic()  //afiseaza graficul prin puncte
{
 // se afiseaza punctele
 ctabpct::cafiseaza();

 // se unesc punctele consecutive prin segmente de dreapta
 struct linesettingstype temp;
 getlinesettings(&temp);
 int c = getcolor();
 setlinestyle(SOLID_LINE,0,NORM_WIDTH);
 int i,j;
 getaspectratio(&i,&j);
 double f=(double)i/j;

 for(i=0;i < n-1;i++) {
    setcolor(ctab[i].retculoare());
```

```
      line(ctab[i].retabs(),f*ctab[i].retord(),
         ctab[i+1].retabs(),f*ctab[i+1].retord());
   }
   setcolor(c);
   setlinestyle(temp.linestyle,temp.upattern,
                temp.thickness);
}

void graficpct::stergegrafic()
/* face invizibil graficul prin puncte */
{
  // se sterg punctele
  ctabpct::csterge();

  // se sterg segmentele de dreapta care unesc punctele consecutive
  struct linesettingstype temp;
  getlinesettings(&temp);
  int c = getcolor();
  setlinestyle(SOLID_LINE,0,NORM_WIDTH);
  setcolor(getbkcolor());
  int i,j;
  getaspectratio(&i,&j);
  double f = (double)i/j;

  for(i=0;i < n-1;i++)
     line(ctab[i].retabs(),f*ctab[i].retord(),
        ctab[i+1].retabs(),f*ctab[i+1].retord());
  setcolor(c);
  setlinestyle(temp.linestyle,temp.upattern,
               temp.thickness);
}
```

31.11 Să se scrie un program care trasează graficul unei funcţii care se defineşte prin puncte.

Acest program este similar cu cel din exerciţiul 31.7. Spre deosebire de acesta, programul de faţă afişează punctele date ca nişte c_puncte, iar punctele consecutive sînt unite prin segmente de dreaptă.

PROGRAMUL BXXXI11

```
#include <conio.h>
#include <stdlib.h>
#include "BXXXI10.CPP"
#include "BXXXI3.CPP"
```

```
main()
/* traseaza graficul unei functii data prin puncte:
        - punctele se afiseaza ca niste c_puncte;
        - punctele consecutive se unesc prin segmente de dreapta.
*/
{
  int nrpuncte = 0;

  if(citInt("numarul punctelor = ",nrpuncte) == 0){
     cout << "S-a tastat EOF\n";
     exit(1);
  }
  if(nrpuncte == 0){
     cout << "Numarul de puncte = 0\n";
     exit(1);
  }
  int gd = DETECT,gm;
  initgraph(&gd,&gm,"c:\\borlandc\\bgi");
  graficpct functie(nrpuncte);
  cleardevice();
  functie.afisgrafic();
  getch();
  closegraph();
}
```

Observaţie:

Propunem cititorului să execute programul de mai sus pentru funcţia dată prin punctele din tabelul de mai jos.

numărul punctelor = 11

Număr curent	Abscisă	Ordonată	Culoare
1	100	60	1
2	200	60	2
3	100	200	3
4	200	200	4
5	100	60	5

(continuare)

Număr curent	Abscisă	Ordonată	Culoare
6	100	200	6
7	150	250	7
8	200	200	8
9	200	60	9
10	150	10	10
11	100	60	11

31.12 Tipul abstract *graficpct* permite trasarea pe ecranul grafic a graficelor funcţiilor definite prin puncte. Abscisa şi valoarea funcţiei, precum şi culoarea punctului sînt date care se citesc de la intrarea standard. Adesea, coordonatele punctelor rezultă în urma unor calcule. Se pot defini tablouri de tip *epunct* sau *cepunct* ale căror elemente se determină prin calcul.

Pentru a trasa graficul unei funcţii definite în acest fel, vom extinde tipul abstract *graficpct*, adăugînd un constructor cu doi parametri:

- *nr*: numărul elementelor de tip *epunct* pentru care se trasează graficul;
- *te*: tabloul de tip *epunct* ale cărui elemente definesc coordonatele celor *nr* puncte ale graficului funcţiei, precum şi culoarea acestora.

Numim *tgrafic* tipul abstract care permite trasarea graficelor funcţiilor fie citind datele de la intrarea standard, fie folosind un tablou de tip *epunct* ale cărui elemente se determină prin calcule.

Clasa *tgrafic* este o clasă derivată a clasei *graficpct*.

Ea are trei constructori:

- constructorul implicit;
- constructor cu un singur parametru, utilizat în cazul cînd datele se citesc de la intrarea standard;
- constructorul cu doi parametri, amintit mai sus.

FIŞIERUL BXXXI12.H

```
#ifndef __GRAFICPCT_H
#include "BXXXI10.CPP"
#define __GRAFICPCT_H
#endif

class tgrafic:public graficpct{
 public:
    tgrafic();
    tgrafic(int nr);
    tgrafic(int nr,epunct *te);
};
```

Observaţie:

Funcţiile membru *afisgrafic* şi *stergegrafic* pot fi apelate pentru obiecte ale clasei *tgrafic* deoarece ele sînt moştenite de această clasă.

Constructorii clasei *tgrafic* se definesc în fişierul de mai jos, de extensie *CPP*.

FIŞIERUL BXXXI12

```
#ifndef __TGRAFIC_H
#include "BXXXI12.H"
#define __TGRAFIC_H
#endif

inline tgrafic::tgrafic():graficpct()
{
}

inline tgrafic::tgrafic(int nr):graficpct(nr)
{
}

tgrafic::tgrafic(int nr,epunct *te)
{
 if(nr <= 0){
    cout << "Numar puncte eronat\n";
    n = 0;
 }
 else{
    n = nr;
    tab = new epunct[nr];
```

```
        if(tab == 0){
            cout << "memorie insuficienta\n";
            n = 0;
        }else{
            int i;
            for(i=0;i < n;i++)
                tab[i]=te[i];
            vtabpct=0;
            (epunct *)ctab=tab;
        }
    }
}
```

31.13 În ultimul timp se încearcă definirea evoluţiei "probabile" a unei persoane folosind diferite metode matematice mai mult sau mai puţin sofisticate. O metodă foarte simplă pretinde că defineşte ascensiunile sau decăderile probabile ale unei persoane în primii 70 de ani de viaţă folosind data naşterii persoanei respective.

Evident, aceste tendinţe pot fi puternic influenţate de condiţiile concrete ale vieţii fiecăruia şi o ascensiune rezultată din metoda pe care o indicăm mai jos, poate să nu se adeverească.

Tendinţele de ascensiune şi coborîre se vor afişa pe ecranul grafic.

Ca date de intrare se dau ziua, luna şi anul naşterii persoanei în cauză.

Ziua şi luna se concatenează împreună pentru a forma un întreg de 4 cifre.

De exemplu, 15 octombrie (luna a 10-a) formează întregul 1510, 25 septembrie (luna a 9-a), formează întregul 2509.

În cazul în care ziua aparţine intervalului [1,9], aceasta se înmulţeşte cu 10. Deci, 4 octombrie formează întregul 4010, 4 septembrie formează întregul 4009 etc.

Numărul obţinut în acest fel se înmulţeşte cu anul naşterii care şi el este un întreg de 4 cifre. Primele 7 cifre ale produsului obţinut în acest fel, reprezintă puncte de ascensiune şi descreştere din viaţa persoanei respective. Acestea aparţin intervalului [0,9] şi ele reprezintă puncte pe ordonată (valorile funcţiei pentru care se trasează graficul). Pe axa absciselor se reprezintă anii de viaţă din 10 în 10.

Prima cifră a produsului corespunde abscisei de valoare 10, a două cifră, abscisei de valoare 20 şi aşa mai departe.

Punctele reprezentate în acest fel se unesc prin segmente de dreaptă.

Conform acestei metode, persoanele născute în zilele de 1, 2 şi 3 ale unei luni calendaristice vor avea acelaşi grafic cu cele născute în zilele de 10, 20 şi respectiv 30 ale aceleeaşi luni calendaristice.

Data calendaristică se citeşte apelînd funcţia *pcit_data_calend*, definită în exerciţiul 8.4. Aceasta apelează funcţiile definite în exerciţiile 8.2 şi 8.3. Data calendaristică se validează de către funcţia *pcit_data_calend* care apelează în acest scop funcţia *v_calend*, definită în exerciţiul 6.5. Această funcţie foloseşte tabloul global *nrzile* care defineşte numărul de zile din lunile calendaristice.

PROGRAMUL BXXXI13

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>

int nrzile[] =
 {0,31,28,31,30,31,30,31,31,30,31,30,31};

#include "BVI5.CPP"
#include "BVIII2.CPP"
#include "BVIII3.CPP"
#include "BVIII4.CPP"
#include "BXXXI12.CPP"

main() /* graficul tendintelor evolutiilor probabile ale unei persoane */
{
 int zi,luna,an;

 if(pcit_data_calend(&zi,&luna,&an) == 0){
    cout << "s-a tastat EOF\n";
    exit(1);
 }
 char zona[5]; /* zona in care se concateneaza ziua cu
                  luna calendaristica */
 char cluna[3];    // luna pe caractere
 itoa(luna,cluna,10);

 if(zi < 10)   // zi in intervalul [1,9]
    zi *= 10;
 itoa(zi,zona,10);
```

```
if(luna < 10)  /* daca luna este in intervalul [1,9], se
                  concateneaza un zero dupa zi */
    strcat(zona,"0");
strcat(zona,cluna);
long zilunan = atol(zona);
/* zilunan are ca valoare întregul format din concatenarea
   zilei cu luna calendaristica */
zilunan *= an;  /* produsul dintre anul nasterii si intregul
                   format din concatenarea zilei cu luna;
                   este un intreg de cel mult 8 cifre */
char zonacif[9];
ltoa(zilunan,zonacif,10);

/* zonacif contine cifrele care reprezinta ordonatele
   functiei de reprezentat pe ecranul grafic */
int gd = DETECT,gm;
initgraph(&gd,&gm,"c:\\borlandc\\bgi");
int ax,ay;  // coeficientii de aspect
getaspectratio(&ax,&ay);
double f=(double)ax/ay;

// afiseaza axele de coordonate
line(40,40*f,40,330*f);        // axa ordonatelor
line(20,310*f,530,310*f);      // axa absciselor
int crtcul=getcolor();         // salveaza culoarea curenta
setcolor(YELLOW);  // defineste culoarea pentru diviziuni

// traseaza diviziunile pe axa ordonatelor
for(int y=1; y < 10;y++){
    char cif[2];
    itoa(y,cif,10);
    int z = 310-30*y;

    // afiseaza ordonata
    outtextxy(20,z*f,cif);

    /* prin punctul de diviziune se duce paralela cu axa absciselor */
    line(30,z*f,530,z*f);
}

// traseaza diviziunile pe axa absciselor
for(int x = 10;x < 80;x += 10){
    char ab[3];
```

```
    itoa(x,ab,10);
    int z = 30+5*x;

    // afiseaza abscisa
    outtextxy(z,340*f,ab);
    // traseaza diviziunea
    line(z+4,310*f,z+4,330*f);


  // reface culoarea curenta
  setcolor(crtcul);
  cepunct hor[7];  // tablou de 7 elemente pentru pastrarea
                   // punctelor functiei de trasat
  int i;
  // se definesc punctele functiei

  for(x=10,i=0;x <= 70;x += 10,i++)
     hor[i] = cepunct(34+5*x,
            (310-30*(zonacif[i]-'0')),LIGHTRED);
  // se instantiaza un obiect de tip tgrafic care se
  // initializeaza cu elementele tabloului hor
  tgrafic functie(7,hor);

  // se traseaza graficul functiei
  functie.afisgrafic();
  getch();
  closegraph();
}
```

31.2. Formatarea în memorie

Aşa cum s-a amintit la începutul capitolului de faţă, din clasa *streambuf* derivă clasa *strstreambuf*. Obiectele acestei clase se folosesc la realizarea operaţiilor de formatare în memorie. Ele sînt utilizate în clase ale ierarhiei de clase care are ca rădăcină clasa *ios*. Astfel, din clasa *ios* derivă clasa *strstreambase*, iar din aceasta derivă alte trei clase: *istrstream, ostrstream* şi *strstream*. Aceste trei clase mai au, fiecare, încă o clasă de bază şi anume:

- clasa *istrstream* derivă din clasa *istream*,
- clasa *ostrstream* derivă din clasa *ostream*,
- clasa *strstream* derivă din clasa *iostream*.

Clasa *ostrstream* se utilizează pentru a realiza conversii din format intern în şiruri de caractere care se păstrează în memorie. Aceste conversii sînt analoge cu cele realizate cu ajutorul funcţiei *sprintf*.

Clasa *ostrstream* are doi constructori de prototip:

```
ostrstream();
```

şi

```
ostrstream(char *zona,int n);
```

unde:

zona - Defineşte adresa zonei de memorie în care se păstrează şirul de caractere rezultat în urma conversiilor datelor din formatul intern.

n - Defineşte dimensiunea zonei de memorie spre care pointează *zona*.

În cazul constructorului fără parametri, se defineşte în mod automat zona de memorie receptoare.

Adresa acestei zone de memorie poate fi cunoscută apelînd funcţia membru *str* de prototip:

```
char *str();
```

Ea returnează adresa zonei tampon alocată în mod automat la utilizarea unui obiect instanţiat fără parametri:

```
ostrstream obiect;
```

Dimensiunea zonei de memorie alocate se obţine prin apelul funcţiei membru *pcount* a clasei *ostrstream* de prototip:

```
int pcount();
```

Clasa *ostrstream*, fiind derivată din clasa *ostream*, moşteneşte supraîncărcările operatorului de *inserare* (<<). Acestea permit ca operatorul de inserare să poată fi utilizat cu instanţieri ale clasei *ostrstream*.

Exemplu:

```
ostrstream obiect;
int i;
...
obiect << i << '\0';
```

Zona de memorie receptoare se alocă automat la execuţia acestei instrucţiuni expresie. De aceea, în continuare se poate apela funcţia *str* pentru a determina adresa zonei de memorie în care se păstrează caracterele

rezultate prin conversia lui *i* din binar în zecimal:

```
char *zona = obiect.str();
```

În continuare avem acces la caracterele din zona receptoare prin intermediul pointerului *zona*.

De exemplu, instrucţiunea:

```
cout << zona;
```

afişează la terminal valoarea lui *i* convertită în zecimal.

Zona de memorie alocată în acest fel trebuie să fie eliberată de utilizator folosind operatorul *delete*:

```
delete zona;
```

Clasa *istrstream* permite realizarea de conversii inverse faţă de clasa *ostrstream*. Aceste conversii sînt similare cu cele obţinute cu ajutorul funcţiei *sscanf*.

Clasa *istrstream* are doi constructori de prototip:

```
istrstream(char *zona);
istrstream(char *zona,int n);
```

unde:

zona - Este pointer spre începutul zonei de memorie care conţine şirul de caractere supus conversiilor.

n - Este dimensiunea zonei de memorie spre începutul căreia pointează zona.

Clasa *istrstream* este derivată din clasa *istream* şi de aceea ea moşteneşte, printre altele, supraîncărcările operatorului de *extragere* (>>). În felul acesta, operatorul de extragere se poate folosi cu obiecte ale clasei *istrstream*.

Clasele amintite mai sus, implică includerea fişierului *strstream.h*.

Exemplu:

Secvenţa de mai jos, utilizată la citirea unui întreg de tip *long* de la intrarea standard:

```
...
for(;;){
 char tab[255];
 printf(text);
 if(gets(tab) == 0)
     exit(1);
```

```
   if(sscanf(tab,"%ld",&x) == 1)
      break;
   printf("Nu s-a tastat un intreg\n");
   printf("Se reia citirea intregului\n");
  }
  ...
```

se poate rescrie ca mai jos, înlocuind funcţiile *gets, sscanf* şi *printf* cu funcţii membru şi operatori supraîncărcaţi în clasele definite în fişierul *strstream.h*.

```
  ...
  for(;;){
   char tab[255];
   cout << text;
   if(cin.getline(tab,255,'\n') == 0)
      exit(1);  // stare de eroare (EOF)
   istrstream obiect(tab);
   if(obiect >> x)
      break;
   cout << "Nu s-a tastat un intreg\n";
   cout << "Se reia citirea intregului\n";
  }
  ...
```

31.3. Prelucrarea fişierelor

În cele ce urmează avem în vedere operaţii de prelucrare a fişierelor înregistrate pe discuri.

La prelucrarea fişierelor se folosesc obiecte ale clasei *filebuf* care este o clasă derivată a clasei *streambuf*.

Obiectele clasei *filebuf* sînt utilizate de clase specifice pentru operaţii cu fişiere şi care aparţin ierarhiei care are ca rădăcină clasa *ios*. În această ierarhie este definită clasa *fstreambase* care este o clasă derivată a clasei *ios*.

Clasa *fstreambase* este o clasă de bază pentru următoarele 3 clase:

- *ifstream* folosită la operaţii de intrare (citire);
- *ofstream* folosită la operaţii de ieşire (scriere);
- *fstream* folosită la operaţii de intrare/ieşire (citire/scriere).

Aceste trei clase, au fiecare, încă o clasă de bază şi anume:

- *ifstream* derivă din clasa istream;

- *ofstream* derivă din clasa ostream;
- *fstream* derivă din clasa iostream.

De aici rezultă că pentru obiectele acestor clase se pot aplica funcţiile membru ale claselor lor de bază, inclusiv supraîncărcările operatorilor de *inserare* şi *extragere*.

Prelucrarea unui fişier de pe disc începe cu *deschiderea* lui. În acest scop se poate proceda în unul din următoarele moduri:

1. Se utilizează funcţia membru *open* cu un obiect al uneia din clasele *ifstream, ofstream* sau *fstream* instanţiat fără parametri.
2. Se utilizează parametri pentru deschiderea fişierului la instanţierea obiectelor.

În primul caz, obiectele se instanţiază folosind constructori impliciţi.

Fie, de exemplu, instanţierea:

```
ifstream obiect;
```

Obiectului *obiect* i se ataşează un fişier concret la un apel al funcţiei *open* de forma:

```
obiect.open(...)
```

Funcţia *open* este definită în clasa *fstreambase* şi este supraîncărcată în clasele derivate din aceasta. Ea are prototipul:

void open(char ****fisier*,int** *mod***,int** *protectie***);**

Parametrul *fisier* este un pointer spre un sir de caractere care defineşte numele şi extensia fişierului care se deschide, precum şi calea spre fişierul respectiv, dacă acesta nu se află în directorul curent.

Parametrul *mod* defineşte modul de deschidere al fişierului. În acest scop se pot folosi enumeratorii definiţi în clasa *ios* astfel:

```
class ios {
 public:
     enum open_mode {
         in=0x01,out=0x02,ate=0x04,app=0x08,
         trunc=0x10,nocreate=0x20,noreplace=0x40,
         binary=0x80
     };
};
```

Aceşti enumeratori au următoarele semnificaţii:

- *in*: fişierul se deschide în intrare (citire);

- *out*: fisierul se deschide în ieşire (scriere);
- *ate*: după deschiderea fişierului, poziţia curentă în fişier este sfîrşitul fişierului; acest mod se utilizează pentru a face căutări în fişier începînd cu sfîrşitul acestuia;
- *app*: deschidere pentru a adăuga înregistrări la sfîrşitul fişierului;
- *trunc*: dacă fişierul există, conţinutul lui se pierde şi se creează un fişier nou; acest mod este implicit cînd este prezentă opţiunea *out* (deschidere în scriere); el este incompatibil cu opţiunile *ate* şi *app*;
- *nocreate*: fişierul care se deschide cu această opţiune trebuie să existe; el nu se deschide în creare;
- *noreplace*: dacă fişierul există şi este prezentă opţiunea *out*, atunci deschiderea este admisă numai dacă se utilizează opţiunea *ate* sau *app*;
- *binary*: fişierul se deschide pentru a fi prelucrat binar; în mod implicit fişierele se consideră că sînt prelucrate pe caractere (mod text);

La o deschidere de fişier se pot folosi împreună mai multe opţiuni, folosind operatorul "|" (sau logic pe biţi).

Pentru obiectele clasei *ifstream*, opţiunea *in* este implicită, iar pentru cele ale clasei *ofstream*, opţiunea *out* este implicită.

Parametrul *protecţie* defineşte modul de acces la fişier. În mod implicit acest parametru are valoarea zero şi aceasta înseamnă că fişierul respectiv nu are restricţii la acces.

Rezultatul operaţiei de deschidere de fişiere poate fi testat cu ajutorul cuvîntului de stare al streamului pentru care s-a apelat funcţia *open*.

Exemple:

1.
```
ifstream fisier;

//se deschide fisierul fis.dat in citire
fisier.open("fis.dat",ios::nocreate);

//se testeaza starea obiectului fisier dupa deschidere
if(fisier){
    //deschidere corecta
    ...
```

```
}
else{
    ...
    cout << "incident la deschiderea lui\
    fis.dat\n";
    ...
}
```

Opţiunea ios::in este implicită.

2. `ofstream fis_iesire;`

```
//se deschide in scriere fisierul factura.dat
fis_iesire.open("factura.dat");

if(!fis_iesire){
   cout << "incident la deschiderea lui\
               factura.dat\n";
   ...
}
```

3. `ofstream fis;`

```
//se deschide pentru adaugare fisierul binar stoc.dat
fis.open("stoc.dat",ios::app | ios::binary);

//se testeaza starea obiectului fis dupa deschidere
if(!fis){
    ...
    cout << "incident la deschiderea lui\
                stoc.dat\n";
    ...
}
else{
    ...
    //deschidere corecta
    ...
}
```

Cea de a doua posibilitate de a deschide un fişier este aceea de a folosi parametrii corespunzători la instanţierea obiectelor claselor *ifstream, ofstream* sau *fstream*. În acest caz, constructorul apelat pentru instanţierea obiectului apelează în mod automat funcţia *open*.

Constructorii care deschid fişierele la instanţierea obiectelor claselor amintite mai sus, au aceeaşi parametri ca şi funcţia *open*:

nume_clasa(**char** **fisier*,**int** *mod*,**int** *protectie*);

unde:

nume_clasa - Este *ifstream, ofstream* sau *fstream*.

Parametrul *protectie* este şi în acest caz un parametru implicit, avînd valoarea implicită egală cu *zero*.

Exemplele de mai sus pot fi rescrise astfel:

1.
```
ifstream fisier("fis.dat",ios::nocreate);
if(fisier){
// deschidere corecta
    ...
}
else{
    ...
    cout << "incident la deschiderea lui\
             fis.dat\n";
    ...
}
```

2.
```
ofstream fis_iesire("factura.dat");
if(!fis_iesire){
    cout << "incident la deschiderea lui\
             factura.dat\n";
    ...
}
```

3.
```
ofstream fis("stoc.dat",ios::app | ios::binary);
if(!fis){
    ...
    cout << "incident la deschiderea lui\
             stoc.dat\n";
    ...
}
else{
    ...
    // deschidere corecta
    ...
}
```

După deschiderea unui fişier se pot face prelucrări asupra fişierului respectiv.

După terminarea prelucrării unui fişier, în conformitate cu modul în care a fost deschis, acesta trebuie închis. Închiderea unui fişier se realizează cu ajutorul funcţiei *close*. Aceasta se apelează în mod obişnuit:

```
obiect.close();
```

sau

```
pobiect -> close();
```

unde:

obiect - Este o instanţiere a uneia din clasele *ifstream, ofstream* sau *fstream*.

În mod analog, *pobiect* este un pointer spre unul din tipurile implementate prin aceste clase.

La prelucrarea fişierelor păstrate pe disc se pot folosi funcţiile membru ale claselor *istream, ostream* şi *iostream*, acestea fiind clase de bază pentru clasele specifice prelucrării fişierelor de pe disc. În particular, se pot folosi operatorii de *inserare* şi *extragere* pentru operaţii de scriere cu format şi respectiv citire cu format dintr-un fişier de pe disc.

Aceşti operatori, precum şi funcţiile membru *get, getline, put, read* şi *write* amintite în paragraful 31.1, pot fi folosite pentru a realiza operaţii cu fişiere pe disc. Cu ajutorul acestora se realizează operaţii de intrare/ieşire cu acces secvenţial.

Accesul aleator se poate realiza folosind funcţii membru specifice ale claselor *ostream* şi *istream*.

Astfel, clasa *ostream* are funcţiile membru *seekp* şi *tellp* care pot fi utilizate la realizarea accesului aleator în scrierea înregistrărilor în fişiere.

În mod anolog, clasa *istream* are funcţiile membru *seekg* şi *tellg* care pot fi utilizate la realizarea accesului aleator la citirea înregistrărilor din fişiere.

Funcţia *tellp* returnează poziţia curentă în streamul curent faţă de începutul lui. Are prototipul:

long tellp();

Funcţia *seekp* realizează poziţionarea în *streamul* curent.

Aceasta are prototipul:

ostream& seekp(long *n,seek_dir rel= beg*);

unde:

seek_dir - Se defineşte în clasa *ios* astfel:

enum *seek_dir*{*beg,cur,end*};

Aceşti parametri se interpretează astfel:

- *n:* deplasament în număr de octeţi;
- *rel*:
  - *beg*: deplasamentul *n*, se consideră faţă de începutul streamului;
  - *cur*: deplasamentul *n*, se consideră faţă de poziţia curentă din stream;
  - *end*: deplasamentul *n*, se consideră faţă de sfîrşitul streamului.

Funcţiile membru ale clasei *istream* au prototipuri şi semnificaţii similare:

long tellg();

şi

istream& seekg(long *n,seek_dir rel=beg*);

Menţionăm că funcţiile *tellp, tellg, seekp şi seekg* se pot utiliza şi în legătură cu obiectele care sînt instanţieri ale claselor *istrstream, ostrstream* şi *strstream*.

O altă funcţie membru utilă este:

gcount

funcţie membru a clasei *istream*;

Ea are prototipul:

int gcount();

Funcţia *gcount* returnează numărul de octeţi citiţi la ultima operaţie de intrare.

Exerciţii:

31.14 Să se scrie un program care copiază intrarea standard la ieşirea standard.

Acest program a mai fost realizat în exerciţiile: 16.1 şi 16.4. În exerciţiul 16.1 s-au folosit funcţiile *read* şi *write* din biblioteca limbajului C şi care au prototipurile în fişierul *io.h*. Ele sînt funcţii de prelucrare, de nivel inferior, ale fişierelor.

În exerciţiul 16.4 se utilizează funcţiile *getc* şi *putc* care au prototipurile

în fişierul *stdio.h*.

În acest exerciţiu se utilizează două variante şi anume:

- utilizarea funcţiilor *get* şi *put*, prima fiind funcţie membru a clasei *istream*, iar cealaltă fiind funcţie membru a clasei *ostream* (această variantă este similară cu cea din exerciţiul 16.4);
- utilizarea operatorilor de *extragere* şi *inserare*.

PROGRAMUL BXXXI14

```
#include <iostream.h>

main()
/* copiaza intrarea standard la iesirea standard folosind functiile
   membru get si put */
{
 char c;
 while(cin.get(c))
     cout.put(c);
}
```

Observaţie:

Programul se termină la întîlnirea sfîrşitului de fişier. În acest moment, streamul *cin* intră în stare de eroare şi referinţa la el, returnată de funcţia *get*, este zero. În felul acesta, ciclul *while* se întrerupe şi odată cu el se întrerupe şi execuţia programului.

PROGRAMUL BXXXI14A

```
#include <iostream.h>
#include <iomanip.h>

main()
/* copiaza intrarea standard la iesirea standard folosind operatorii de
   extragere si inserare */
{
 char c;
 while(cin >> resetiosflags(ios::skipws) >> c)
     cout << c;
}
```

Observaţii:

1. Programul se termină la întîlnirea sfîrşitului de fişier. În acest moment, streamul *cin* intră în stare de eroare. Operatorul > > este supraîncărcat în aşa fel încît să returneze o referinţă la *streamul* la care se aplică.

Această referinţă este egală cu zero dacă *streamul* respectiv se află în stare de eroare.

2. Aceste programe pot fi folosite pentru a copia fişiere de tip text de pe disc sau pe disc, precum şi pe imprimantă. În acest scop se vor face redirectări corespunzătoare.

Exemple:

a. Afişarea pe display a conţinutului fişierului *fis.dat* de pe disc:

   >BXXXI14A < fis.dat

b. Afişarea pe imprimanta paralelă a conţinutului fişierului *fis.dat* de pe disc:

   >BXXXI14A < fis.dat > prn

31.15 Să se scrie un program care citeşte de la intrarea standard datele ale căror formate sînt definite mai jos şi le scrie în fişierul *misc.dat* din directorul curent.

Fişierul este organizat binar, fiecare articol conţinînd următoarele date:

tip	denumire	um	cod	pret	cantitate
I	REZISTENTA	010KO	123456789	10	10000

Acest program este similar cu cel definit în exerciţiul 16.6.

PROGRAMUL BXXXI15

```
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
#define MAX 50

typedef struct {
    char tip[2];
    char den[MAX+1];
    int val;
    char unit[3];
    long cod;
    float pret;
    float cant;
```

```
} ARTICOL;
```

/* 6 articole in zona tampon */

```
union {
    ARTICOL a[6];
    char zt[6*sizeof(ARTICOL)];
} buf;

int citart(ARTICOL *str)
```

/* - citeste datele de la intrarea standard si le pastreaza in zona spre care pointeaza str;
- returneaza valoarea zero la intilnirea sfirsitului de fisier si unu in caz contrar. */

```
{
 char t[255];
 float x,y;

 for(;;) {
    cout << "tip den um cod pret cant\n";
    if(cin >> setw(2)>>str -> tip >> setw(50) >>
      str -> den >> str -> val >> setw(3) >>
        str -> unit>> str -> cod >> x >> y){
         str -> pret = x;
         str -> cant = y;
          return 1;
      }

    // streamul este in stare de eroare
    if(cin.eof())
        return 0;  //EOF

    // nu este sfirsit de fisier
    // se anuleaza starea de eroare
    cin.clear();

    // se avanseaza peste eventualele caractere existente dupa eroare
    cin.getline(t,255,'\n');
    cout << "Date eronate; se reia citirea\n";
 }  // sfirsit for
} // sfirsit functie citart
```

```
main()
/* creeaza fisierul misc.dat cu datele citite de la intrarea standard */
{
 // se deschide misc.dat in creare binara
 ofstream misc("misc.dat",ios::binary);
 int n;
 if(!misc){
     cout << "Nu se poate deschide fisierul\
             misc.dat in creare\n";
     exit(1);
 }
 // se creaza fisierul misc.dat
 for(;;){
     // se umple zona tampon a fisierului
     for(int i=0;i < 6;i++)
         if((n=citart(&buf.a[i])) == 0)
             break;   // s-a intilnit EOF
     // se scrie zona tampon daca nu este vida
     if(i){   // zona tampon nu este vida
         misc.write(buf.zt,i*sizeof(ARTICOL));
         if(!misc){   // eroare la scriere in fisier
             cout << "Eroare la scrierea in fisierul\
                     misc.dat\n";
             exit(1);
         }
     }
     if(n == 0)
         break;   // s-a intilnit EOF
 }   // sfirsit for
 misc.close();
}  // sfirsit main
```

31.16 Să se scrie un program care afişează articolele păstrate în fişierul *misc.dat* creat în exerciţiul precedent, pentru tip = I.

Amintim că *tip* este o literă:

- I - intrări;
- E - ieşiri;
- T - transfer etc.

PROGRAMUL BXXXI16

```
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
#define MAX 50

typedef struct {
    char tip[2];
    char den[MAX+1];
    int val;
    char unit[3];
    long cod;
    float pret;
    float cant;
} ARTICOL;

union {
    ARTICOL a[6];
    char zt[6*sizeof(ARTICOL)];
}buf;

main()
/* citeste si afiseaza articolele din fisierul misc.dat pentru tip = I */
{
 ifstream misc("misc.dat",ios::binary);
 int n;

 if(!misc) {
    cout << "Eroare la deschiderea fisierului\
           misc.dat in citire\n";
    exit(1);
 }

 // se citeste fisierul misc.dat si se listeaza articolele pentru tip = I
 cout << "\n\n\n" << setw(30) << "INTRARI" <<
         "\n\n\n";
 do {
    misc.read(buf.zt,6*sizeof(ARTICOL));

   // calculeaza numarul articolelor citite
    n = misc.gcount()/sizeof(ARTICOL);
    for(int i=0;i < n;i++)
       if(buf.a[i].tip[0] == 'I'){
```

```
            cout.setf(ios::left,ios::adjustfield);
            cout << setw(50) << buf.a[i].den;
            cout.setf(ios::right,ios::adjustfield);
            cout << setw(3) << setfill('0') <<
                     buf.a[i].val;
            cout.fill(' ');
            cout << buf.a[i].unit;
            cout << " " << setw(9) << setfill('0')
                 << buf.a[i].cod;
            cout.fill(' ');
            cout << " " << buf.a[i].pret;
            cout << " " << buf.a[i].cant << "\n";
        }  // sfirsit for
  } while(misc);
  if(!misc.eof())
      cout << "incident la citirea din fisierul\
               misc.dat\n";
  misc.close();
}
```

31.17 Să se scrie un program care citeşte şiruri de caractere de la intrarea standard şi le transferă într-un fişier pe disc.

Numele fişierului este *textmen.txt*.

PROGRAMUL BXXXI17

```
#include <fstream.h>
#include <stdlib.h>

main()
/* - citeste texte de la intrarea standard si le pastreaza pe disc in
     fisierul textmen.txt;
   - programul se termina la intilnirea sfirsitului de fisier. */
{
 // se deschide fisierul textmen.txt in creare

 ofstream textmen("textmen.txt");
 char buf[255];

 for(;;){
 // citeste o linie de la intrare
     if(cin.getline(buf,255,'\n') == 0)
         break;    // sfirsitul de fisier
```

```
        // se scrie linia in fisier
        for(int i=0;buf[i];i++)
            textmen.put(buf[i]);
        textmen.put('\n');
    }
    textmen.close();
}
```

31.18 Să se definească tipul abstract *bara* care permite implementarea unei *bare de meniuri*.

O bară de meniuri este un dreptunghi de lăţimea unei linii de pe ecranul setat în mod text. În acest dreptunghi sînt afişate texte care pot fi selectate cu ajutorul tastelor cu săgeţi (săgeata spre stînga şi spre dreapta). Aceste texte le numim *meniuri*.

Fondul pentru bara de meniuri este afişat folosind culoarea definită de constanta LIGHTGRAY.

Textele se scriu folosind culoarea definită de constanta BLACK.

Unul dintre meniuri este meniul *curent*. Pentru acesta, fondul se afişează folosind culoarea definită de constanta CYAN.

Meniul curent se *activează* prin acţionarea tastei *Enter*.

Fiecărui meniu i se ataşează o funcţie care se apelează în momentul *activării* meniului respectiv. Numim *acţiuni* aceste funcţii.

Programul aplicativ conţine un tablou global cu pointeri spre aceste acţiuni, numit *tabact*. Acţiunile au prototipul:

```
int actiune(void);
```

Interfaţa dintre acţiuni se realizează cu ajutorul variabilelor de tip global.

Pentru a putea gestiona simplu acţiunile programului, se introduce tipul PF astfel:

```
typedef int (*PF)();
```

Tabloul *tabct* se declară astfel:

```
int (*tabact[])() = {act1,act2,...,actn};
```

unde:

act1, act2, ..., actn - Sînt numele acţiunilor.

Tipul abstract *bara* are trei date membru:

- *tab*;
- *n*;
- *crt*.

Data membru *tab* este un tablou de elemente de tip *meniu*. Aceste elemente se păstrează în memoria *heap*.

Data membru *n* defineşte numărul elementelor tabloului *tab*. El este egal cu numărul elementelor tabloului *tabact*.

Data membru *crt* defineşte meniul curent.

Meniurile se consideră numerotate de la unu.

La afişarea barei de meniuri, în mod implicit, meniul cel mai din stînga este selectat (crt=1).

Funcţiile membru ale tipului *bara* sînt:

- *bara* (constructor);
- *afisbara*;
- *activbara*;
- *apelact*;
- *execmeniu*.

Constructorul clasei *bara* rezervă zonă de memorie în memoria *heap* pentru tabloul *tab*. Apoi, citeşte textele de meniu şi păstrează în memoria *heap* elementele de tip *meniu* ale tabloului *tab*.

Textele de meniu se citesc de la intrarea standard.

Funcţia *afisbara* afişează bara de meniuri în linia unu a ecranului.

Funcţia *activbara* afişează meniul selectat, adică al crt-lea meniu.

Funcţia *apelact* se apelează la acţionarea tastei *Enter*. Ea are ca efect apelul acţiunii corespunzătoare meniului al crt-lea.

Funcţia *execmeniu* citeşte de la tastatură caracterele utilizate pentru gestiunea barei de meniuri (tastele cu săgeţi, tasta *Enter* şi tasta *Esc*) şi le interpretează în mod corespunzător:

- săgeţile permit selectarea meniului curent;
- tasta *Enter* permite apelul acţiunii corespunzătoare meniului curent;
- tasta *Esc* termină execuţia aplicaţiei sub controlul barei de meniuri.

Menţionăm că o aplicaţie se poate executa sub controlul mai multor bare de meniuri.

Tipul abstract *meniu* conţine următoarele date membru:

- *text*;

şi

- *pf*.

Data membru *text* este un pointer spre un text de meniu.

Data membru *pf* este un pointer de tip *PF*. Are ca valoare adresa acţiunii ataşate meniului.

Funcţiile membru ale tipului *meniu* sînt:

- 2 constructori;
- *afis*;
- *apelact*;
- *retlung*.

Clasa *meniu* are un constructor implicit vid şi unul care permite iniţializarea unui obiect de tip *meniu* cu textul meniului şi cu adresa acţiunii meniului.

Textul meniului se păstrează în memoria *heap*.

Funcţia *afis* afişează textul meniului.

Funcţia *apelact* apelează acţiunea corespunzătoare meniului.

Funcţia *retlung* returnează lungimea textului meniului în număr de caractere.

Clasele *meniu* şi *bara* se definesc în fişierul de extensie *h* de mai jos, iar funcţiile membru ale lor, în fişierul de extensie *cpp*.

FIŞIERUL BXXXI18.H

```
typedef int(*PF)();

class meniu {
    char *text;  // pointer spre textul meniului
    PF pf;       // pointer spre actiunea meniului
 public:
    meniu();
    meniu(char *t,PF f);
```

```
        void afis(); //afiseaza textul meniului
        int apelact(); //apeleaza actiunea meniului
        unsigned retlung(); //returneaza lungimea textului
                                      //meniului
    };

    class bara{
        meniu *tab; //tabloul de meniuri
        int n;      //numarul elementelor lui tab
        int crt;    //indicele meniului curent
      public:
        bara(int s=1); //s defineste indicele meniului curent
        void afisbara(); //afiseaza bara de meniuri
        void activbara();//afiseaza meniul curent
        int apelact();    //apeleaza actiunea meniului curent
        void execmeniu(); //gestioneaza evenimentele
                          //de la tastatura
    };
```

FIŞIERUL BXXXI18

```
    #include <string.h>
    #include <iostream.h>
    #ifndef __BARA
    #include "bxxxi18.h"
    #define __BARA
    #endif

    meniu::meniu()  //constructor vid
    {
    }

    meniu::meniu(char *t,PF f)
    //constructor utilizat pentru definirea barei de meniuri
    {
      //rezerva zona pentru textul meniului
     if((text=new char[strlen(t)+1])==0){
        cout << "memorie insuficienta\n";
        exit(1);
     }
      //transfera textul meniului in memoria heap
     strcpy(text,t);
```

```
 // initializeaza pointerul spre actiunea meniului
 pf=f;
}

unsigned meniu::retlung()
// returneaza lungimea textului meniului
{
 return strlen(text);
}

void meniu::afis() // afiseaza textul meniului
{
 cprintf("  %s ",text);
}

bara::bara(int a) // constructor
// a este indexul meniului curent
{
 crt=a;

 int nr=sizeof tabact/sizeof(PF);
 // determina numarul meniurilor

 if((tab=new meniu[nr])==0)
 // rezerva zona pentru bara de meniuri
 {
    cout << "memorie insuficienta\n";
    exit(1);
 }

 for(n=0;n<nr;n++)
// citeste textele meniurilor si initializeaza adresa actiunilor meniurilor
  {
    char t[255];
    cout << "textul meniului curent:";
    if(!(cin >> t))
        break;
    tab[n]= meniu(t,tabact[n]);
 }
}
```

```
void bara::afisbara() //afiseaza bara de meniuri
{
 //ascunde cursor
 _AH=1;
 _CH=0x20;
 geninterrupt(0x10);
 clrscr();
 int dim=2;

 for(int i=0;i<n;i++)    //determina lungimea barei de meniuri
     dim +=tab[i].retlung()+3;
 window(1,1,dim,1); //fereastra pentru bara de meniuri

 textbackground(LIGHTGRAY); //fondul barei
 textcolor(BLACK);      //culoarea de afisare a meniurilor
 clrscr();
 for(i=0;i<n;i++){      //afisarea meniurilor
     tab[i].afis();
 }
}

void bara::activbara() //afiseaza meniul curent
{
 int dim=2;

 for(int i=0;i<n;i++) {
     if(crt==i+1)
         break;
     dim +=tab[i].retlung()+3;
 }
 window(dim,1,dim+tab[i].retlung()+3,1);
 textbackground(CYAN);
 textcolor(BLACK);
 clrscr();
 tab[i].afis();
}

void bara::execmeniu() //functia de gestiune a evenimentelor
{
 char t[255];
 cin.getline(t,255,'\n');   //videaza buferul de intrare
 char c=0;
```

```
for(;c!=27;){   //se iese la Esc
    c=getch();
    switch(c){
        case 0:                  //caracter de control
            c=getch();
            switch(c){
                case 75:  //caracterul <-
                    if(crt==1)
                        break;
                    crt--; //cursorul pe meniul precedent
                    afisbara();
                    activbara();
                    break;
                case 77:  //caracterul ->
                    if(crt==n)
                        break;
                    crt++;   //cursorul pe meniul urmator
                    afisbara();
                    activbara();
            }
            break;
        case 13:                 //caracterul Enter
            apelact();
            break;
        case 27:                 //caracterul Esc
            break;
    }
  }
}

int bara::apelact()
{
 int i;

 i=tab[crt-1].apelact();
 return i;
}

int meniu::apelact()
{
 int i;
```

```
    i=(*pf)();
    return i;
}
```

31.19 Să se scrie un program care realizează următoarele transformări:

- măsurile ariilor din hectare în jugăre;

și

- măsurile unghiurilor din radiani în grade sexagesimale.

Programul folosește o bară de meniuri cu următoarele texte:

HaJug RadSexagesimal

La activarea meniului *HaJug* se apelează acțiunea *hajug* care realizează conversia din hectare în jugăre.

La activarea meniului *RadSexagesimal* se apelează acțiunea *radsexa* care realizează conversia din radiani în grade sexagesimale.

Ambele acțiuni apelează funcția *citd* care afișează un text și apoi citește un număr de la intrarea standard. Numărul respectiv este interpretat ca număr de hectare, dacă funcția care face apelul este *hajug* și ca grade în radiani dacă apelul se face de către funcția *radsexa*.

Rezultatul conversiei se atribuie variabilei globale *rezultat*.

În ambele cazuri, se apelează funcția *afrez* care afișează un text, împreună cu valoarea variabilei *rezultat*.

Acțiunile *hajug, radsexa* și funcțiile *citd* și *afrez*, împreună cu funcția principală sînt definite în fișierul de mai jos, de extensie *cpp*.

În acest fișier se mai definesc datele globale:

- *rezultat*;

și

- *tabact*.

Funcția principală instanțiază obiectul *bm*:

bara bm;

La această instanțiere se apelează constructorul clasei *bara* care citește textele meniurilor (*HaJug* și *RadSexagesimal*) și inițializează tabloul *tab* cu textele respective și adresele acțiunilor. Apoi se apelează funcția *afisbara* pentru a afișa bara de meniuri definită de obiectul *bm*. Apelînd în continuare funcția *activbara* pentru același obiect, se afișează meniul

curent al cărui fond are culoarea definită de constanta CYAN. După aceea se apelează funcţia *execmeniu* pentru obiectul *bm*. În acest moment programul se află în aşteptarea tastării unui caracter.

Se pot acţiona tastele cu săgeţi (săgeata spre stînga şi săgeata spre dreapta), tasta *Enter* şi *Esc*.

Alte taste dacă sînt acţionate nu au nici un efect.

Cu ajutorul tastelor cu săgeţi se poate defini meniul curent.

La acţionarea tastei *Enter* se va apela una din acţiunile *hajug* sau *radsexa*. La acţionarea tastei *Esc*, se revine din funcţia *exemeniu* şi se întrerupe execuţia programului.

FIŞIERUL BXXXI19

```
#include <stdlib.h>
#include <conio.h>
#include <dos.h>

int hajug();     // conversie din ha in jugar
int radsexa();   // conversie din radian in sexagesimal
int (*tabact[])()={hajug,radsexa}; // tabela de actiuni

#include "bxxxi18.cpp"        // clasele meniu si bara
#include <strstream.h>

double rezultat;

int citd(char *text,double& d)
/* - afiseaza text;
  - citeste un numar;
  - returneaza:
        0 - la EOF;
        1 - altfel. */
{
 char t[255];
 double x;
 // afiseaza cursor

 _AH=1;
 _CH=6;
 _CL=7;
 geninterrupt(0x10);
 window(1,2,80,25);
```

```
  textbackground(BLACK);
  textcolor(WHITE);
  clrscr();

  for(;;){
     gotoxy(1,wherey()+1);

     if(text && *text)
        cout << text;

     if(cin.getline(t,255,'\n')==0){
      // ascunde cursor
        _AH=1;
        _CH=0x20;
        geninterrupt(0x10);
        return 0;
     }

     istrstream ob(t);
     if(ob >> x)
        break;
     cout << "nu s-a tastat un numar\n";
  }
  d=x;

  // ascunde cursor
  _AH=1;
  _CH=0x20;
  geninterrupt(0x10);
  return 1;
}

void afrez(char *s)
{
 gotoxy(1,wherey()+1);
 cout << s << " " << rezultat;
}

int hajug()
{
 if(citd("aria in Ha:",rezultat)==0) {
     cout << "S-a tastat EOF\n";
     return 0;
```

```
 }
 rezultat /= 0.57546415;
 afrez("Aria in Jugare");
 return 1;
}

int radsexa()
{
 if(citd("masura unghiului in radiani:",
        rezultat)==0) {
    cout << "S-a tastat EOF\n";
    return 0;
 }
 rezultat *= 180.0/3.14159265358979;
 afrez("Unghiul in Grade sexagesimale");
 return 1;
}

main(){
// sterge tot ecranul

 window(1,1,80,25);
 textbackground(BLACK);
 textcolor(WHITE);
 clrscr();
 bara bm;   // instantiaza bara de meniuri
 bm.afisbara();
 bm.activbara();
 bm.execmeniu();

// afiseaza cursor
 _AH=1;
 _CH=6;
 _CL=7;

 geninterrupt(0x10);
 window(1,1,80,25);
 textbackground(BLACK);
 textcolor(WHITE);
 clrscr();
}
```

BIBLIOGRAFIE

1. **C.Bohn, G.Jacopini**

   FlowDiagrams, Turning Machines and Languages with OnlyTwo Formation Rules

   Comm ACM, 9, 5, pag. 366 - 371, 1966

2. **Brian W. Kernighan, Dennis M. Richie**

   The C Programming Language

   Prentice-Hall, Inc, Englewood Cliffs

   New Jersey 1978

3. **Donald E. Knuth**

   Tratat de programarea calculatoarelor, vol. 1 - 3

   Editura Tehnică

   Bucureşti 1974

4. **Valentin Cristea şi alţii**

   Dicţionar de informatică

   Editura ştiinţifică şi enciclopedică

   Bucureşti 1981

5. **Narian Gehani**

   C: An Advanced Introduction

   AT&T Bell Laboratories Murray Hill

   New Jersey 1985

6. **Joel E. Richardson, Michel J. Carey, Daniel T. Schuh**

   The Design of the E Programming Language

   Computer Sciences Department

   University of Wisconsin Madison

Wi53706 1989

7. * * *
BORLAND INTERNATIONAL TURBO C++:
Getting Started
1990

8. * * *
BORLAND INTERNATIONAL TURBO C++:
Programmer's Guide
1990

9. **Frederic Lung**
Le Langage C++
CNIT Paris - La Defense
1990

10. **Setrag Khoshafian, Razmik Abnous**
Object Orientation Concepts, Languages, Databases, User Interfaces
John Wiley&Sons, Inc. New York
Chichester Brisbone Toronto
Singapore 1990

11. **Aaron M. Tenenbaum, Yedidyah Langsam, Moshe J. Augenstein**
Data Structures Using C
Prentice - Hall International, Inc.
1990

12. **Clara Ionescu, Ioan Zsako**
Structuri arborescente cu aplicaţiile lor
Editura Tehnică

București 1990

13. **Liviu Negrescu**
Introducere în limbajul C vol. 1 - 2
Colecția GLOB
1990

14. **Stroustrup Bjarne**
The C++ Programming Language
Second Edition
Copyright 1991 by AT&T
Bell Telephone Laboratories, Incorporated

15. **Liviu Negrescu**
Limbajul C - Culegere de programe
Fasciculele 1 - 2
Cluj-Napoca 1991

16. **Liviu Negrescu**
Limbajul TURBO C
Editura LIBRIS
Cluj-Napoca 1992

17. **Liviu Negrescu**
Introducere în mediul de dezvoltare integrat TURBO C
Editura LIBRIS
Cluj-Napoca 1992

18. **Ionuț Mușlea**
Inițiere în C++ - Programare orientată pe obiecte
Editura MicroInformatica

Cluj-Napoca 1992

19. **Donald L. Shell**
CACM 2(iulie), pag. 30-32
1959

20. **C. A. R. Hoare**
Quicksort
Comp. j., 5, No. 1
1962

21. **Liviu Negrescu**
Introducere în limbajul C
Editura MicroInformatica
Cluj-Napoca 1993



I.S.B.N. 973-96980-4-2

9880 lei



str. Observatorului nr.1, bl.OS1
3400 Cluj-Napoca
of. PTTR Cluj-Napoca, CP 186
tel / fax 064.198263, 123125



Cartea se adresează unui cerc larg de cititori care doresc să se inițieze în programarea și utilizarea limbajelor C și C++ . Primele două volume realizează o introducere elementară în aceste limbaje.Volumul trei conține programe diverse în C și C++ cu aplicații la rezolvarea problemelor științifice și tehnico-inginerești, în prelucrări de date , precum și în scrierea programelor de sistem.